https://github.com/firmianay/Life-long-Learner/blob/master/linux-kernel-development/chapter-10.md

1)

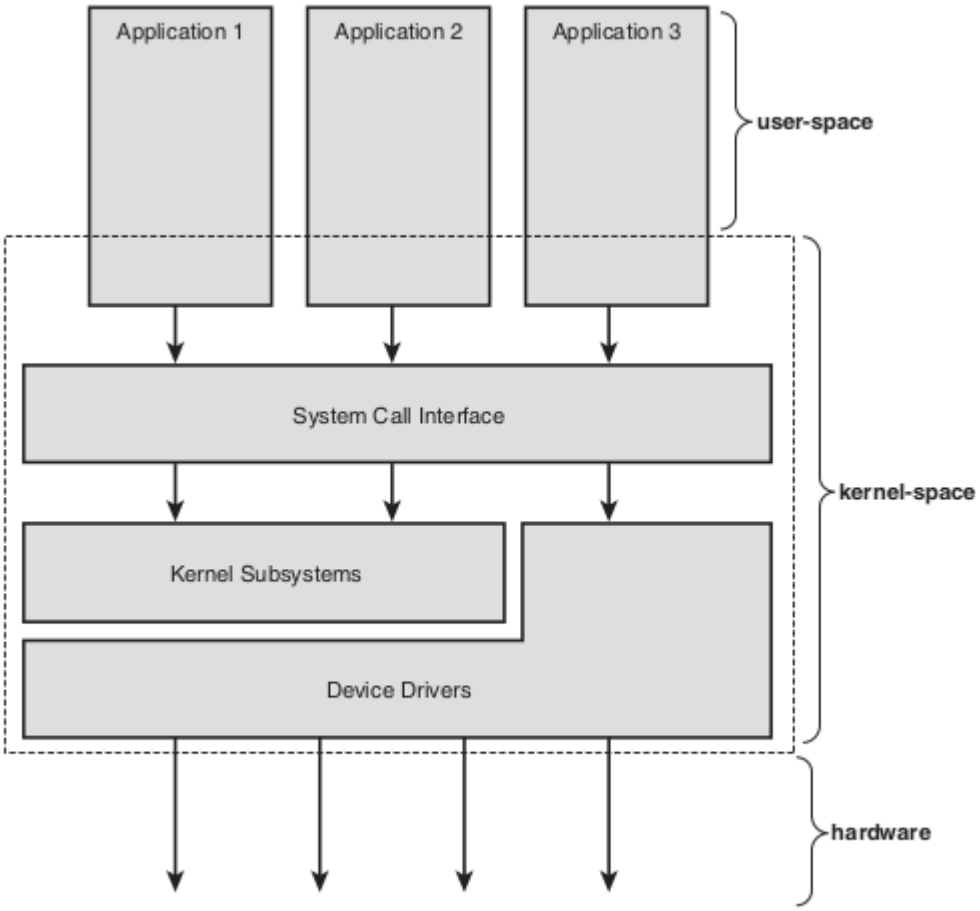

Figure 1.1 Relationship between applications, the kernel, and hardware.

An application typically calls functions in a library -- for example, the C library -- that in turn rely on the system call interface to instruct the kernel to carry out tasks on the application's behalf.

2)

The Kernel Source Tree

Directory	Description
arch	Architecture-specific source
block	Block I/O layer
crypto	Crypto API
Documentation	Kernel source documentation

Directory	Description
drivers	Device drivers
firmware	Device firmware needed to use certain drivers
fs	The VFS and the individual filesystems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
samples	Sample, demonstrative code
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)
tools	Tools helpful for developing Linux
virt	Virtualization infrastructure

The kernel cannot easily execute floating-point operations. Unlike a user-space application, the kernel is not linked against the standard C library, or any other library. The primary reason is speed and size.

Many of the usual libc functions are implemented inside the kernel. For example, the common string manipulation functions are the lib/string.c. Just include the header file <linux/string.h> and have at them.

When a user-space application attempts an illegal memory access, the kernel can trap the error, send the SIGSEGV signal, and kill the process. If the kernel attempts an illegal memory access, however, the results are less controlled. Memory violations in the kernel result in an oops, which is a major kernel error.

Small, Fixed-Size Stack: User-space has a large stack that can dynamically grow. The kernel stack is neither large nor dynamic; it is small and fixed in size. The exact size of the kernel's stack varies by architecture.

3)

A process includes a set of resources such as open files and pending signals, internal kernel data, processor state, a memory address space with one or more memory mappings, one or more threads of execution, and a data section containing global variables.

Whereas, each thread includes a unique program counter, process stack, and set of processor registers. The kernel schedules individual threads, not processes. Linux does not differentiate between threads and processes; a thread is just a special kind of process.

virtualized processor: The virtual processor gives the process the illusion that it alone monopolizes the system, despite possibly sharing the processor among hundreds of other processes.

virtual memory: It lets the process allocate and manage memory as if it alone owned all the memory in the system.

fork() system call creates a new process by duplicating an existing one. The parent resumes execution and the child starts execution at the same place: where the call to fork() returns. The fork() system call returns from the kernel twice: once in the parent process and again in the newborn child.

The exec() family of function calls creates a new address space and loads a new program into it.
The exit() system call exits a program. This function terminates the process and free all its resources. When a process exits, it is placed into a special zombie state that represents terminated processes until the parent call wait() or waitpid().

The kernel stores the list of processes in a circular doubly linked list called the task list. Each element in the task list is a process descriptor of the type struct task_struct, which is defined in <linux/sched.h>. The process descriptor contains all the information about a specific process.

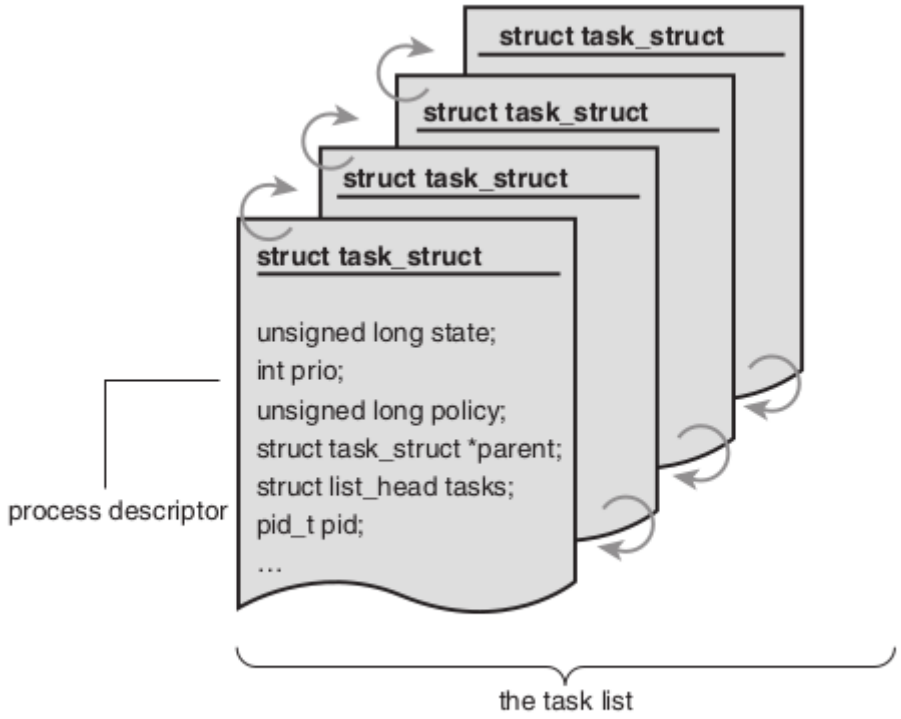

Figure 3.1 The process descriptor and task list.

The task_struct is allocate via the slab allocator to provide object reuse and cache colouring. The new structure, struct thread_info, lives at the bottom of the stack (for stacks that grow down) and at the top of the stack (for stacks that grow up).

```
struct thread_info {
   struct task_struct    *task;
   struct exec_domain    *exec_domain;
   __u32                 flags;
   __u32                 status;
   __u32                 cpu;
   int                   preempt_count;
   mm_segment_t          addr_limit;
   struct restart_block  restart_block;
   void                  *sysenter_return;
   int                   uaccess_err;
};
```

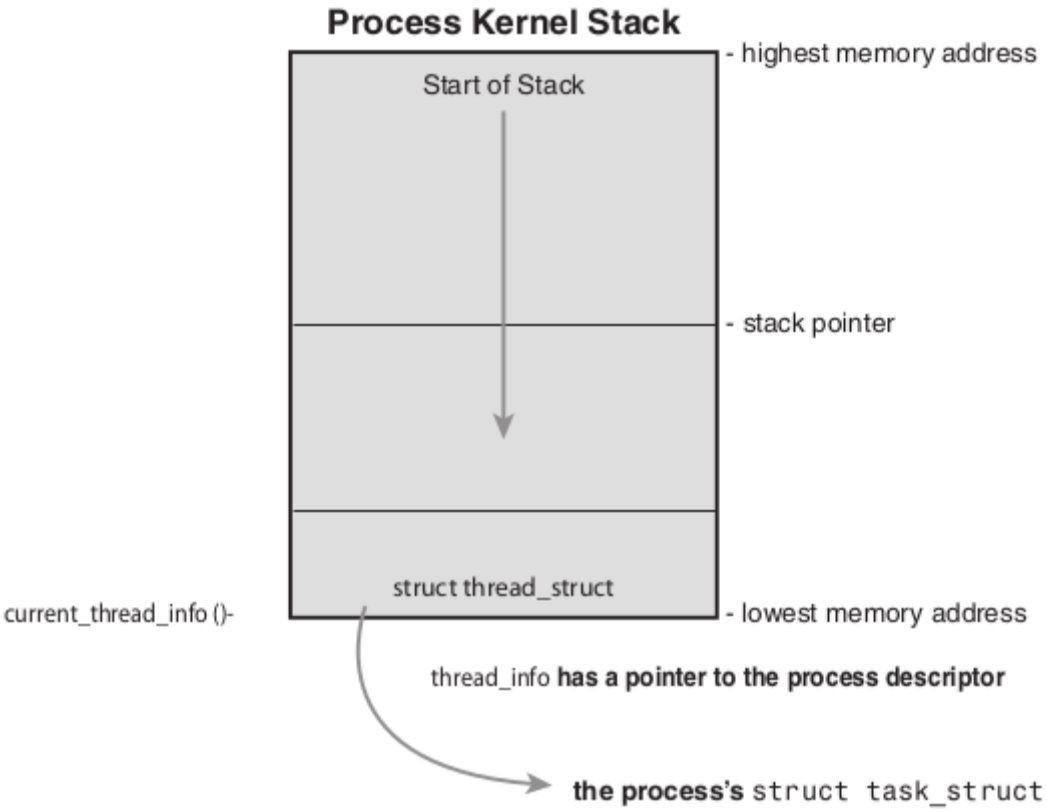

Figure 3.2 The process descriptor and kernel stack.

The system identifies processes by a unique process identification value or (PID) default maximum value is only 32768. (controlled in <linux/threads.h>). Inside the kernel, tasks are typically referenced directly by a pointer to their task_struct structure. Consequently, it is useful to be able to quickly look up the process descriptor of the currently executing task, which is done via the current macro.

The state field of the process descriptor describes the current condition of the process:

- TASK_RUNNING: The process is runnable; it is either currently running or on a runqueue waiting to run. This is the only possible state for a process executing in user-space; it can also apply to a process in kernel-space that is actively running.
- TASK_INTERRUPTIBLE: The process is sleeping (blocked), waiting for some condition to exist. The process also awakes prematurely and becomes runnable if it receives a signal.
- TASK_UNINTERRUPTIBLE: This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal. This is used in situations where the process must wait without interruption or when the event is expected to occur quite quickly.
- __TASK_TRACED: The process is being traced by another process, such as a debugger, via ptrace.
- __TASK_STOPPED: Process execution has stopped; the task is not running nor is it eligible to run. This occurs if the task receives the SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal or if it receives any signal while it is being debugged.

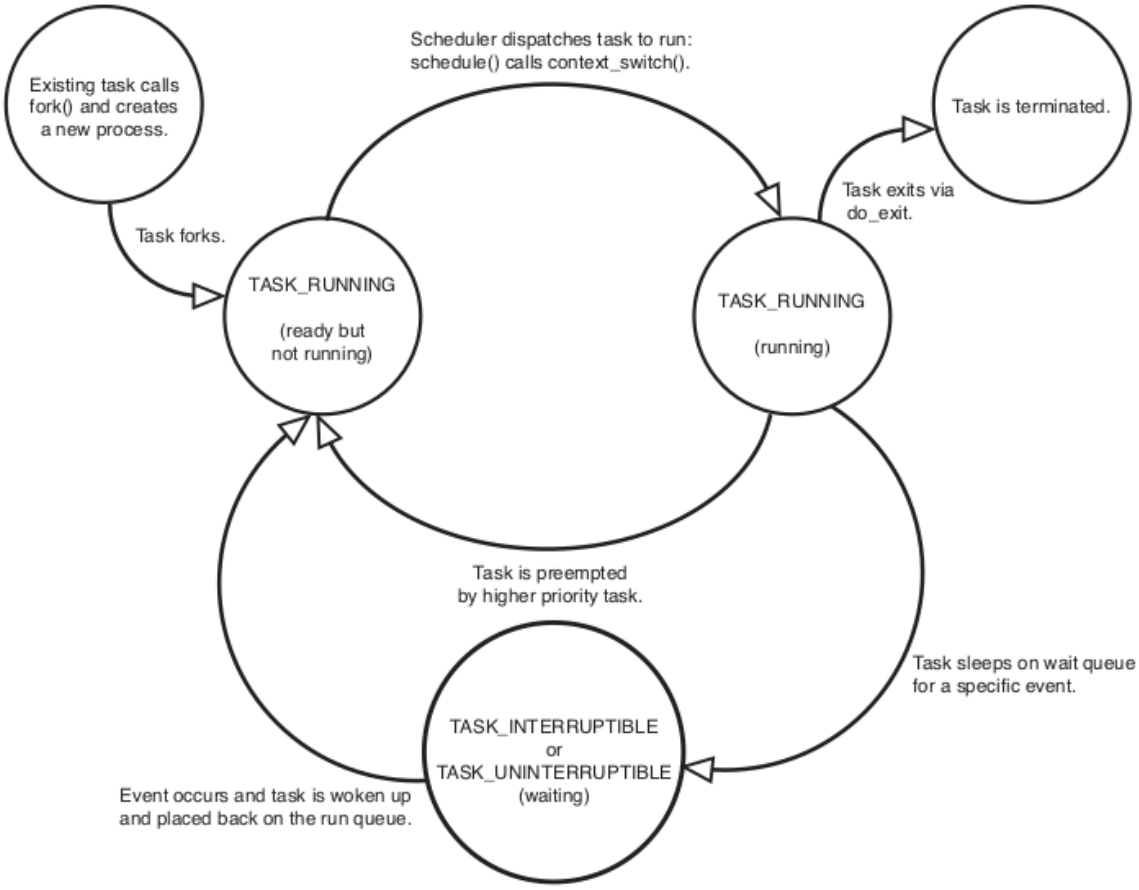

Figure 3.3 Flow chart of process states.

Kernel code often needs to change a process's state. set_task_state(task, state); /* set task 'task' to state 'state' */

The code is read in from executable file and executed within the program's address space. Normal program execution occurs in user-space. When a program executes a system call or triggers an exception, it enters kernel-space. At this point, the kernel is said to be “executing on behalf of the process” and is in process context.

All processes are descendants of the init process, whose PID is one. The kernel starts init in the last step of the boot process. The init process reads the system initscripts and executes more programs, eventually completing the boot process.

- parent: Every process on the system has exactly one parent.
- children: Every process has zero or more children.
- siblings: Processes that are all direct children of the same parent are called siblings.

The relationship between processes is stored in the process descriptor. Each task_struct has a pointer to the parent's task_struct, named parent, and a list of children, named children. To obtain the process descriptor of current process's parent:

struct task_struct *my_parent = current->parent

To iterate over a process's children:

```
struct task_struct *task;

struct list_head *list;


list_for_each(list, &current->children) {

   task = list_entry(list, struct task_struct, sibling);

   /* task now points to one of current's children */

}
```

- fork(): Creates a child process that is a copy of the current task.
- exec(): Loads a new executable into the address space and begins executing it.

fork() is implemented through the use of copy-on-write pages. Copy-on-write (COW) is a technique to delay or altogether prevent copying of the data. Rather than duplicate the process address space, the parent and the child share a single copy. The duplication of resources occurs only when they are written. The only overhead incurred by fork() is the duplication of the parent's page tables and the creation of a unique process descriptor for the child.

Linux implements fork() via the clone() system call which takes a series of flags that specify which resources the parent and child process should share. The clone() system call calls do_fork().

The vfork() system call has the same effect as fork(), except that the page table entries of the parent process are not copied. The child executes as the sole thread in the parent's address space, and the parent is blocked until the child either calls exec() or exits. The child is not allowed to write to the address space.

Linux implements all threads as standard processes. A thread is merely a process that shares certain resources with other process. Each thread has a unique task_struct and appears to the kernel as a normal process. Threads just happen to share resources, such as an address space, with other processes.

Threads are created the same as normal tasks, with the exception that the clone() system call is passed flags corresponding to the specific resources to be shared:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

kernel threads: Standard processes that exist solely in kernel-space. Kernel threads do not have an address space, they operate only in kernel-space and do not context switch into user-space.

Linux delegates several tasks to kernel threads, most notably the flush tasks and the ksoftirqd task (kcompactd, kswapd etc. ?). Kernel threads are created on system boot by other kernel threads. Indeed, a kernel thread can be created only by another kernel thread. The interfaces, declared in <linux/kthread.h>, for spawning a new kernel thread from an existing one is:

```
struct task_struct *kthread_create(int (*threadfn)(void *data), void *data, const char namefmt[], ...)
```

The above process is created in an unrunnable state, it will not start running until explicitly woken up via``wake_up_process()`.

Cleaning up after a process and removing its process descriptor are separate steps. This enables the If a parent exits before its children, any of its child tasks must be reparented to a new process. The code attempts to find and return another task in the process's thread group. If another task is not in the thread group, it finds and returns the init process. Now that a suitable new parent for the children is found, each child needs to be located and reparented to reaper.

When a task is ptraced, it is temporarily reparented to the debugging process. When the task's parent exits, however, it must be reparented along with its other siblings. The init process routinely calls wait() on its children, cleaning up any zombies assigned to it.

4)

O(1) scheduler's eventual replacement in kernel version 2.6.23, the Completely Fair Scheduler (CFS).

Processes can be classified as either I/O-bound or processor-bound.

- I/O-bound processes: spend much of its time submitting and waiting on I/O requests.(eg. GUI)
- processor-bound processes: spend much of their time executing code.(eg. MATLAB)

The scheduling policy in a system must attempt to satisfy two conflicting goals: fast process response time (low latency) and maximal system utilization (high throughput). Linux, aiming to provide good interactive response and desktop performance, optimizes for process response (low latency), thus favouring I/O-bound processes over processor-bound processes.

A common type of scheduling algorithm is priority-based scheduling. The Linux kernel implements two separate priority ranges:

- nice value: a number from -20 to +19 with a default of 0. Larger nice values correspond to a lower priority. In Linux, it is a control over the proportion of time slice.
- real-time priority: configurable values that by default range from 0 to 99. Higher real-time priority values correspond to a greater priority.

The time slice is the numeric value that represents how long a task can run until it is pre-empted. Linux's CFS scheduler does not directly assign time slices to processes, but assigns processes a proportion of the processor. Under the new CFS scheduler, the decision is a function of how much of a proportion of the processor the newly runnable processor has consumed. If it has consumed a smaller proportion of the processor than the currently executing process, it runs immediately, pre-empting the current process. If not, it is scheduled to run at a later time.

The Linux scheduler is modular, enabling different algorithms to schedule different types of processes. This modularity is called scheduler classes. The base scheduler code, which is defined in kernel/sched.c, iterates over each scheduler class in order of priority. The highest priority scheduler class that has a runnable process wins, selecting who runs next. The Completely Fair Scheduler (CFS) is the registered scheduler class for normal processes, called SCHED_NORMAL in Linux. CFS is defined in kernel/sched_fair.c.

CFS is based on a simple concept: Model process scheduling as if the system had an ideal, perfectly multitasking processor. Put generally, the proportion of processor time that any process receives is determined only by the relative difference in niceness between it and the other runnable processes. The nice values, instead of yielding additive increases to time slices, yield geometric differences. The absolute time slice allotted any nice value is not an absolute number, but a given proportion of the

processor. CFS is called a fair scheduler because it gives each process a fair share (a proportion) of the processor's time.

The vruntime variable stores the virtual runtime of a process, which is the actual runtime (the amount of time spent running) normalized (or weighted) by the number of runnable processes. The virtual runtime's units are nanoseconds and therefore vruntime is decoupled from the timer tick.

The core of CFS's scheduling algorithm: Pick the task with the smallest vruntime. CFS uses a red-black tree to manage the list of runnable processes and efficiently find the process with the smallest vruntime. A red-black tree, called a rbtree in Linux, is a type of self-balancing binary search tree.

The process that CFS wants to run next, which is the process with the smallest vruntime, is the leftmost node in the tree. CFS's process selection algorithm is thus summed up as "run the process represented by the leftmost node in the rbtree."

CFS adds processes to the rbtree and caches the leftmost node, when a process becomes runnable (wakes up) or is first created via fork(). Adding processes to the tree is performed by enqueue_entity. This function updates the runtime and other statistics and then invokes __enqueue_entity() to perform the actual heavy lifting of inserting the entry into the red-black tree.

This function traverses the tree in the while() loop to search for a matching key (inserted process's vruntime). It moves to the left child if the key is smaller than the current node's key and to the right child if the key is larger. If it ever moves to the right, even once, it knows the inserted process cannot be the new leftmost node, and it sets leftmost to zero. If it moves only to the left, leftmost remains one, and we have a new leftmost node and can update the cache by setting rb_leftmost to the inserted process. When out of the loop, the function calls rb_link_node() on the parent node, making the inserted process the new child. The function rb_insert_color() updates the self-balancing properties of the tree.

CFS removes processes from the red-black tree when a process blocks (becomes unrunnable) or terminates (ceases to exist).

The main entry point into the process schedule is the function schedule(), defined in kernel/sched.c. schedule() is generic to scheduler classes. It finds the highest priority scheduler class with a runnable process and asks it what to run next. The only important part of the function is its invocation of pick_next_task(), defined in kernel/sched.c, which goes through each scheduler class, starting with the highest priority, and selects the highest priority process in the highest priority.

Sleeping: The task marks itself as sleeping, puts itself on a wait queue, removes itself from the red-black tree of runnable, and calls schedule() to select a new process to execute.

Waking up: The task is set as runnable, removed from the wait queue, and added back to the red-black tree.

A wait queue is a simple list of processes waiting for an event to occur. Wait queues are represented in the kernel by wake_queue_head_t. They are created statically via DECLARE_WAITQUEUE() or dynamically via init_waitqueue_head(). Processes put themselves on a wait queue and mark themselves not runnable. When the event associated with the wait queue occurs, the processes on the queue are awakened.

Waking is handled via wake_up(), which wakes up all the tasks waiting on the given wait queue. It calls try_to_wake_up(), which sets the task's state to TASK_RUNNING, calls enqueue_task() to add

the task to the red-black tree, and set need_reached if the awakened task's priority is higher than the priority of the current task.

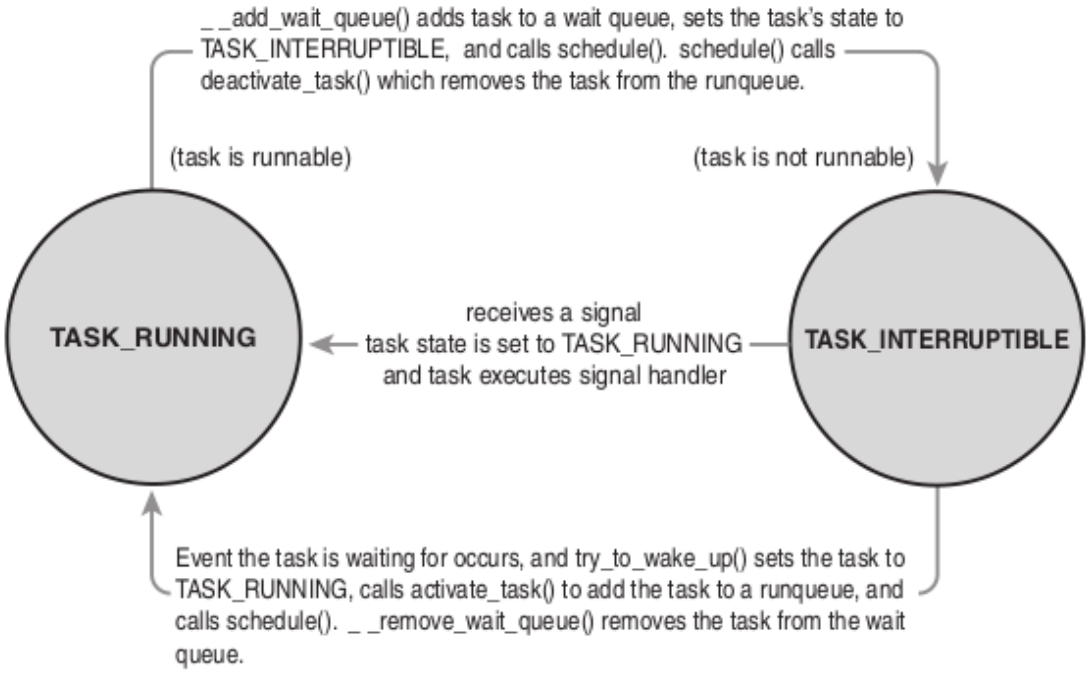

Figure 4.1 Sleeping and waking up.

User pre-emption can occur:

- When returning to user-space from a system call
- When returning to user-space from an interrupt handler

Kernel Pre-emption

The Linux kernel is a fully pre-emptive kernel. It can pre-empt a task running in the kernel so long as it does not hold a lock. To support kernel pre-emption, preempt_count was added to each process's thread_info. This counter begins at zero and increments once for each lock that is acquired and decrements once for each lock that is released. When the counter is zero, the kernel is preemptible. kernel pre-emption can occur:

- When an interrupt handler exits, before returning to kernel-space
- When kernel code becomes preemptible again
- If a task in the kernel explicitly calls schedule()
- If a task in the kernel blocks (which results in a call to schedule())

Linux provides two real-time scheduling policies: SCHED_FIFO and SCHED_RR. The normal, not real-time scheduling policy is SCHED_NORMAL. SCHED_FIFO implements a simple first-in, first-out scheduling algorithm without time slices. SCHED_RR is SCHED_FIFO with time slices. It is a real-time, round-robin scheduling algorithm. Both real-time scheduling policies implement static priorities.

5)

System calls provide a layer between the hardware and user-space processes, which serves three primary purposes:

- Providing an abstracted hardware interface for user space.
- Ensuring system security and stability.
- A single common layer between user-space and the rest of the system allows for the virtualized system provided to processes.

The relationship between a POSIX API, the C library, and system calls:

Figure 5.1 The relationship between applications, the C library, and the kernel with a call to **printf()**.

In Linux, each system call is assigned a unique syscall number that is used to reference a specific system call. the process does not refer to the syscall by name. system call table, stored in sys_call_table, on x86-64 it is defined in arch/x86/kernel/syscall_64.c.

System calls in Linux are faster than in many other operating systems, because of Linux's fast context switch times: entering and exiting the kernel is a streamlined and simple affair.

It is not possible for user-space applications to execute kernel code directly because the kernel exists in a protected memory space. The mechanism to signal the kernel is a software interrupt: Incur an exception, and the system will switch to kernel mode and execute the exception handler which is actually the system call handler. The defined software interrupt on x86 is interrupt number 128, which is incurred via the int $0x80 instruction. The system call handler is the aptly named function system_call().

Simply entering kernel-space alone is not sufficient: the system call number must be passed into the kernel. On x86, the syscall number is passed to the kernel via the eax register.
The system_call() function checks the validity of the given system call number by comparing it to NR_syscalls. If it is larger than or equal to NR_syscalls, the function returns -ENOSYS. Otherwise, the specified system call is invoked:

```
call *sys_call_table(,%rax,8)
```

Because each element in the system call table is 64 bits (8 bytes), the kernel multiplies the given system call number by eight to arrive at its location in the system call table. On x86-32, the code is similar, with the 8 replaced by 4.

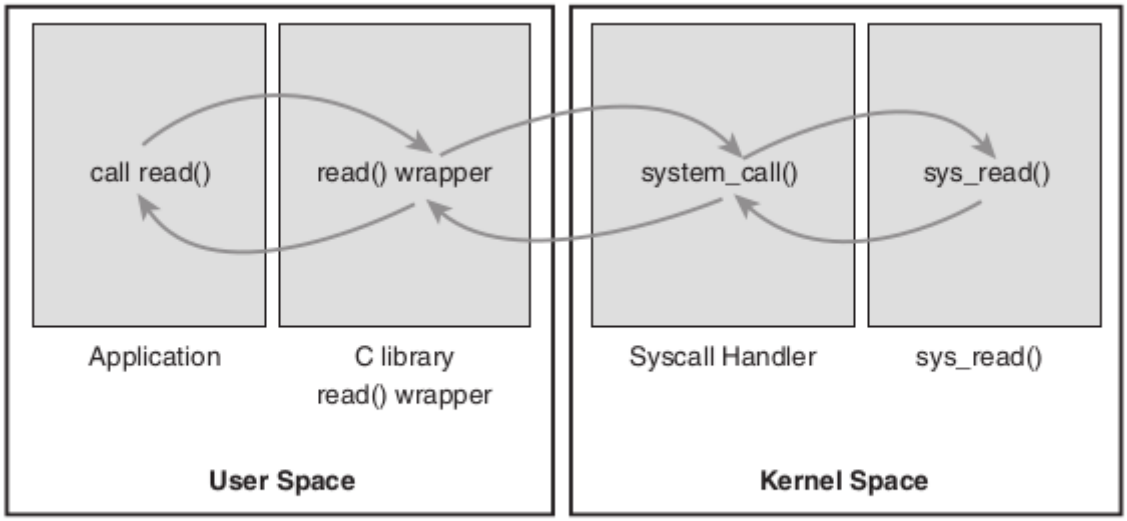

Figure 5.2 Invoking the system call handler and executing a system call.

most syscalls require that one or more parameters be passed to them. The parameters are stored in registers. On x86-32, the registers ebx, ecx, edx, esi, and edi contain, in order, the first five arguments. In the unlikely case of six or more arguments, a single register is used to hold a pointer to user-space where all the parameters are stored.

The return value is sent to user-space also via register. On x86, it is written into the eax register.

The first step in implementing a system call is defining its purpose and the syscall should have exactly one purpose. Multiplexing syscalls (a single system call that does wildly different things depending on a flag argument) is discouraged in Linux.

ioctl() is an example of what not to do. The system call should have a clean and simple interface with the smallest number of arguments possible. They must not change.

Before following a pointer into user-space, the system must ensure that:

- The pointer points to a region of memory in user-space. not in kernel-space on their behalf or reading someone else's data.
- The process must not be able to bypass memory access restrictions. If reading, the memory is marked readable. If writing, the memory is marked writable. If executing, the memory is marked executable.

The kernel provides two methods for performing the requisite checks and the desired copy to and from user-space:

- copy_to_user(): Writing into user-space. It takes three parameters, The first is the destination memory address in the process's address space. The second is the source pointer in kernel-space. The third argument is the size in bytes of the data to copy.
- copy_from_user(): Reading from user-space. The function reads from the second parameter into the first parameter the number of bytes specified in the third parameter.

The kernel is in process context during the execution of a system call. The current pointer points to the current task, which is the process that issued the syscall. In process context, the kernel is capable

of sleeping and is fully preemptible. When the system call returns, control continues in system_call(), which ultimately switches to user-space and continues the execution of the user process.

It is trivial to register an official system call after it is written:

1. Add an entry to the end of the system call table (for most architectures, entry.S). This needs to be done for each architecture that supports the system call. The position of the syscall in the table, starting at zero, is its system call number.
2. For each supported architecture, define the syscall number in <asm/unistd.h>.
3. Compile the syscall into the kernel image (as opposed to compiling as a module). This can be as simple as putting the system call in a relevant file in kernel/, such as sys.c, which is home to miscellaneous system calls.

Accessing the System Call from User-Space: e.g. consider the system call open(), defined as:

```
long open(const char *filename, int flags, int mode)
```

The syscall macro to use this system call without explicit library support would be:

```
#define __NR_open 5

_syscall3(long, open, const char *, filename, int, flags, int, mode)
```

The application can simply call open().

For each macro, there are 2 + 2 × n parameters.The first parameter corresponds to the return type of the syscall. The second is the name of the system call. Next follows the type and name for each parameter in order of the system call. The __NR_open define is in <asm/unistd.h>; it is the system call number. The _syscall3 macro expands into a C function with inline assembly; the assembly performs the steps discussed in the previous section to push the system call number and parameters into the correct registers and issue the software interrupt to trap into the kernel.

System call cons:

- You need a syscall number, which needs to be officially assigned to you.
- After the system call is in a stable series kernel, it is written in stone. The interface cannot change without breaking user-space applications.
- Each architecture needs to separately register the system call and support it.
- System calls are not easily used from scripts and cannot be accessed directly from the filesystem.
- Because you need an assigned syscall number, it is hard to maintain and use a system call outside of the master kernel tree.
- For simple exchanges of information, a system call is overkill.

The alternatives:

- Implement a device node and read() and write() to it. Use ioctl() to manipulate specific settings or retrieve specific information.

- Certain interfaces, such as semaphores, can be represented as file descriptors and manipulated as such.
- Add the information as a file to the appropriate location in sysfs.

6)

The kernel provides a family of routines to manipulate linked lists.

The relationship between a key and its value is called a mapping.

What Data Structure to Use, When

- linked-list: If your primary access method is iterating over all your data.
- queue: If your code follows the producer/consumer pattern, use a queue, particularly if you want (or can cope with) a fixed-size buffer.
- map: If you need to map a UID to an object.
- red-black tree: If you need to store a large amount of data and look it up efficiently.

7)

- Interrupts: asynchronous interrupts generated by hardware.
- Exceptions: synchronous interrupts generated by the processor.

The function the kernel runs in response to a specific interrupt is called an interrupt handler or interrupt service routine (ISR). Each device that generates interrupts has an associated interrupt handler. The interrupt handler for a device is part of the device's driver. They run in a special context called interrupt context.

The processing of interrupts is split into two halves:

- top half: The interrupt handler is the top half. The top half is run immediately upon receipt of the interrupt and performs only the work that is time-critical.
- bottom half: Work that can be performed later is deferred until the bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled.

Drivers can register an interrupt handler and enable a given interrupt line for handling with the function request_irq(), which is declared in <linux/interrupt.h>:

/* request_irq: allocate a given interrupt line */

int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev)

The first parameter, irq, specifies the interrupt number to allocate. The second parameter, handler, is a function pointer to the actual interrupt handler that services this interrupt. This function is invoked whenever the operating system receives the interrupt.

typedef irqreturn_t (*irq_handler_t)(int, void *);

Note the specific prototype of the handler function: It takes two parameters and has a return value of irqreturn_t. The third parameter, flags, can be either zero or a bit mask of one or more of the flags defined in <linux/interrupt.h>. The most important of these flags are:

- IRQF_DISABLED: When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler. When unset, interrupt handlers run with all interrupts except their own enabled. Most interrupt handlers do not set this flag, as disabling all interrupts is bad form.
- IRQF_SAMPLE_RANDOM: This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy.
- IRQF_TIMER: This flag specifies that this handler processes interrupts for the system timer.
- IRQF_SHARED: This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line.

The fourth parameter, name, is name of the device associated with the interrupt. These text names are used by /proc/irq and /proc/interrupts.

The fifth parameter, dev, is used for shared interrupt lines. When an interrupt handler is freed, dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared. This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure. This pointer is unique and might be useful to have within the handlers.

request_irq() returns zero on success and nonzero value indicates an error, in which case the specified interrupt handler was not registered. A common error is -EBUSY, which denotes that the given interrupt line is already in use.

Note that request_irq() can sleep and therefore cannot be called from interrupt context or other situations where code cannot block. On registration, an entry corresponding to the interrupt is created in /proc/irq. The function proc_mkdir() creates new procfs entries. This function calls proc_create() to set up the new procfs entries, which in turn calls kmalloc() to allocate memory.

Freeing an Interrupt Handler: When your driver unloads, you need to unregister your interrupt handler and potentially disable the interrupt line.

```
void free_irq(unsigned int irq, void *dev)
```

If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via dev is removed, but the interrupt line is disabled only when the last handler is removed. With shared interrupt lines, a unique cookie is required to differentiate between the multiple handlers that can exist on a single line and enable free_irq() to remove only the correct handler. In either case, if dev is non-NULL, it must match the desired handler. A call to free_irq() must be made from process context.

```
static irqreturn_t intr_handler(int irq, void *dev)
```

Note that this declaration matches the prototype of the handler argument given to request_irq(). The first parameter, irq, is the numeric value of the interrupt line the handler is servicing. The second parameter, dev, is a generic pointer to the same dev that was given to request_irq() when the

interrupt handler was registered. If this value is unique, it can act as a cookie to differentiate between multiple devices potentially using the same interrupt handler.

The return value of an interrupt handler is the special type irqreturn_t, which has two special values:

- IRQ_NONE: returned when the interrupt handler detects an interrupt for which its device was not the originator.
- IRQ_HANDLED: returned if the interrupt handler was correctly invoked, and its device caused the interrupt.

Alternatively, IRQ_RETVAL(val) may be used. If val is nonzero, this macro returns IRQ_HANDLED. Otherwise, the macro returns IRQ_NONE. These special values are used to let the kernel know whether devices are issuing spurious (unrequested) interrupts. If all the interrupt handlers on a given interrupt line return IRQ_NONE, then the kernel can detect the problem. The return type irqreturn_t which is simply an int. The interrupt handler is normally static because it is never called directly from another file.

A shared handler is similar to a nonshared handler, but has three main differences:

- The IRQF_SHARED flag must be set in the flags argument to request_irq().
- The dev argument must be unique to each registered handler. A pointer to any per-device structure is sufficient; a common choice is the device structure as it is both unique and potentially useful to the handler. It cannot be NULL for a shared handler.
- The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt, which requires both hardware support and associated logic in the interrupt handler; otherwise, the interrupt handler would not know whether its associated device or some other device sharing the line caused the interrupt.

When the kernel receives an interrupt, it invokes sequentially each registered handler on the line. Therefore, it is important that the handler be capable of distinguishing whether it generated a given interrupt. The handler must quickly exit if its associated device did not generate the interrupt. This requires the hardware device to have a status register (or similar mechanism) that the handler can check.

When executing an interrupt handler, the kernel is in interrupt context. Interrupt context cannot sleep, and you cannot call certain functions from interrupt context. The interrupt handler has interrupted other code, so interrupt context is time-critical. The setup of an interrupt handler’s stacks is a configuration option.

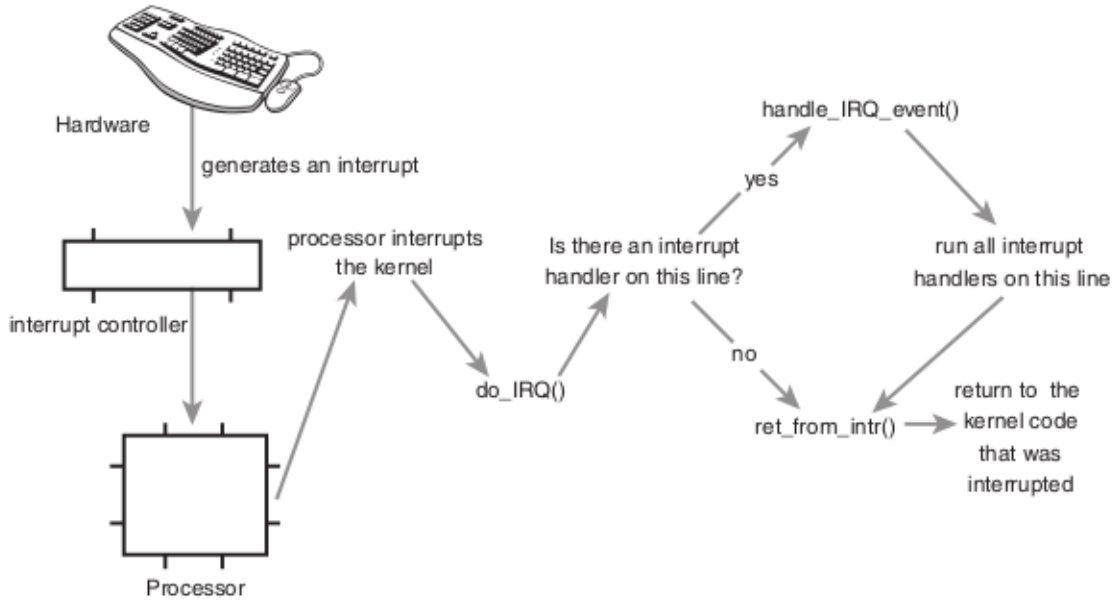

Figure 7.1 The path that an interrupt takes from hardware and on through the kernel.

A device issues an interrupt by sending an electric signal over its bus to the interrupt controller. If the interrupt line is enabled, the interrupt controller sends the interrupt to the processor. Unless interrupts are disabled in the processor, the processor immediately stops what it is doing, disables the interrupt system, and jumps to a predefined location in memory and executes the code located there. This predefined point is set up by the kernel and is the entry point for interrupt handlers.

The interrupt's journey in the kernel begins at this predefined entry point. For each interrupt line, the processor jumps to a unique location in memory and executes the code located there. In this manner, the kernel knows the IRQ number of the incoming interrupt. The initial entry point simply saves this value and stores the current register values on the stack; then the kernel calls do_IRQ().

```
unsigned int do_IRQ(struct pt_regs regs)
```

After the interrupt line is calculated, do_IRQ() acknowledges the receipt of the interrupt and disables interrupt delivery on the line. Next, do_IRQ() ensures that a valid handler is registered on the line and that it is enabled and not currently executing. If so, it calls handle_IRQ_event(), defined in kernel/irq/handler.c, to run the installed interrupt handlers for the line.

1. Since the processor disabled interrupts, they are turned back on if IRQF_DISABLED was not specified during the handler's registration.
2. Each potential handler is executed in a loop. If this line is not shared, the loop terminates after the first iteration. Otherwise, all handlers are executed.
3. add_interrupt_randomness() is called if IRQF_SAMPLE_RANDOM was specified during registration. This function uses the timing of the interrupt to generate entropy for the random number generator.
4. Interrupts are again disabled (do_IRQ() expects them still to be disabled) and the function returns.

Back in do_IRQ(), the function cleans up and returns to the initial entry point, which then jumps to ret_from_intr(). The routine ret_from_intr() is written in assembly. This routine checks whether a reschedule is pending. If a reschedule is pending, and the kernel is returning to user-space, schedule() is called. If the kernel is returning to kernel-space, schedule() is called only if the preempt_count is zero. Otherwise it is not safe to preempt the kernel. After schedule() returns, or if there is no work pending, the initial registers are restored and the kernel resumes whatever was interrupted.

/proc/interrupts

Procfs is a virtual filesystem that exists only in kernel memory and is typically mounted at /proc. Its size is 0, verify using ls -l. Reading or writing files in procfs invokes kernel functions that simulate reading or writing from a real file. The /proc/interrupts file is populated with statistics related to interrupts on the system.

The first column is the interrupt line. The second column is a counter of the number of interrupts received. The third column is the interrupt controller handling this interrupt. The last column is the device associated with this interrupt.

By disabling interrupts, you can guarantee that an interrupt handler will not preempt your current code or disables kernel preemption. since it can't provide any protection from concurrent access from another processor, kernel code more generally needs to obtain some sort of lock to prevent another processor from accessing shared data simultaneously.

These locks are often obtained in conjunction with disabling local interrupts.

local_irq_disable();

/* interrupts are disabled .. */

local_irq_enable();

However, a common concern is a to restore interrupts to a previous state. It is much safer to save the state of the interrupt system before disabling it. Then, when you are ready to reenable interrupts, you simply restore them to their original state:

unsigned long flags;

local_irq_save(flags); /* interrupts are now disabled */

/* ... */

local_irq_restore(flags); /* interrupts are restored to their previous state */

In some cases, it is useful to disable only a specific interrupt line for the entire system. This is called masking out an interrupt line. To disable delivery of a devices' interrupts before manipulating its state, Linux provides four interfaces for this task:

void disable_irq(unsigned int irq);

void disable_irq_nosync(unsigned int irq);

void enable_irq(unsigned int irq);

void synchronize_irq(unsigned int irq);

The first two functions disable a given interrupt line in the interrupt controller. This disables delivery of the given interrupt to all processors in the system. Additionally, the disable_irq() function does not return until any currently executing handler completes. Thus, callers are assured not only that new interrupts will not be delivered on the given line, but also that any already executing handlers have exited.

The function disable_irq_nosync() does not wait for current handlers to complete.

The function synchronize_irq() waits for a specific interrupt handler to exit, if it is executing, before returning.

For each call to disable_irq() or disable_irq_nosync() on a given interrupt line, a corresponding call to enable_irq() is required. Only on the last call to enable_irq() is the interrupt line actually enabled.

The macro irqs_disabled(), defined in <asm/system.h>, returns nonzero if the interrupt system on the local processor is disabled. Otherwise, it returns zero.

Two macros, defined in <linux/hardirq.h>, provide an interface to check the kernel's current context:

in_interrupt()

in_irq()

The first function in_interrupt() returns nonzero if the kernel is performing any type of interrupt handling, including either executing an interrupt handler or a bottom half handler. The macro in_irq() returns nonzero only if the kernel is specifically executing an interrupt handler.

Function	**Description**
local_irq_disable()	Disables local interrupt delivery
local_irq_enable()	Enables local interrupt delivery
local_irq_save()	Saves the current state of local interrupt delivery and then disables it
local_irq_restore()	Restores local interrupt delivery to the given state
disable_irq()	Disables the given interrupt line and ensures no handler on the line is executing before returning
disable_irq_nosync()	Disables the given interrupt line
enable_irq()	Enables the given interrupt line
irqs_disabled()	Returns nonzero if local interrupt delivery is disabled; otherwise returns zero
in_interrupt()	Returns nonzero if in interrupt context and zero if in process context

Function	Description
in_irq()	Returns nonzero if currently executing an interrupt handler and zero otherwise

8)

The job of bottom halves is to perform any interrupt-related work not performed by the interrupt handler. Since interrupt handlers run asynchronously, with at least the current interrupt line disabled, minimizing their duration is important. The following useful tips help decide how to divide the work between the top and bottom half:

- If the work is time sensitive, perform it in the interrupt handler.
- If the work is related to the hardware, perform it in the interrupt handler.
- If the work needs to ensure that another interrupt (particularly the same interrupt) does not interrupt it, perform it in the interrupt handler.
- For everything else, consider performing the work in the bottom half.

Minimizing the time spent with interrupts disabled is important for system response and performance. In the current three methods that exist for deferring work, tasklets are built on softirqs and work queues are their own subsystem.

Softirqs

Softirqs are rarely used directly; tasklets, which are built on softirqs are a much more common form of bottom half. The softirq code lives in the file kernel/softirq.c in the kernel source tree.

Implementing Softirqs

Softirqs are statically allocated at compile time. Unlike tasklets, you cannot dynamically register and destroy softirqs. Softirqs are represented by the softirq_action structure, which is defined in <linux/interrupt.h>:

```
struct softirq_action
{
        void    (*action)(struct softirq_action *);
};
```

A 32-entry array of this structure is declared in kernel/softirq.c:

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

Each registered softirq consumes one entry in the array. Consequently, there are NR_SOFTIRQS registered softirqs. The number of registered softirqs is statically determined at compile time and cannot be changed dynamically. The kernel enforces a limit of 32 registered softirqs.

The Softirq Handler

The prototype of a softirq handler, action, looks like:

```
void softirq_handler(struct softirq_action *);
```

When the kernel runs a softirq handler, it executes this action function with a pointer to the corresponding softirq_action structure as its argument. A softirq never pre-empts another softirq.

Executing Softirqs

A registered softirq must be marked before it will execute. This is called raising the softirq. Usually, an interrupt handler marks its softirq for execution before returning. Pending softirqs are checked for and executed in the following places:

- In the return from hardware interrupt code path
- In the ksoftirqd kernel thread
- In any code that explicitly checks for and executes pending softirqs, such as the networking subsystem

Softirq execution occurs in __do_softirq(), which is invoked by do_softirq(). If there are pending softirqs, __do_softirq() loops over each one, invoking its handler. The following code is a simplified variant of the important part of __do_softirq():

```
u32 pending;

pending = local_softirq_pending();

if (pending) {
   struct softirq_action *h;

   /* reset the pending bitmask */

   set_softirq_pending(0);

   h = softirq_vec;

   do {
      if (pending & 1)
         h->action(h);

      h++;

      pending >>= 1;

   } while (pending);

}
```

It checks for, and executes, any pending softirqs. It specifically does the following:

1. It sets the pending local variable to the value returned by the local_softirq_pending() macro.
2. After the pending bitmask of softirqs is saved, it clears the actual bitmask.
3. The pointer h is set to the first entry in the softirq_vec.
4. If the first bit in pending is set, h->action(h) is called.

5. The pointer h is incremented by one so that it now points to the second entry in the softirq_vec array.
6. The bitmask pending is right-shifted by one.
7. The pointer h now points to the second entry in the array, and the pending bitmask now has the second bit as the first. Repeat the previous steps.
8. Continue repeating until pending is zero, at which point there are no more pending softirqs and the work is done.

Softirqs are reserved for the most timing-critical and important bottom-half processing on the system. Currently, only two subsystems directly use softirqs: networking and block devices. Additionally, kernel timers and tasklets are built on top of softirqs.

Softirqs are declared statically at compile time via an enum in <linux/interrupt.h>. The kernel uses this index, which starts at zero, as a relative priority. Softirqs with the lowest numerical priority execute before those with a higher numerical priority.

The following table contains a list of the existing tasklet types.

Tasklet	**Priority**	**Softirq Description**
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
TASKLET_SOFTIRQ	5	Normal priority tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_SOFTIRQ	7	High-resolution timers
RCU_SOFTIRQ	8	RCU locking

Next, the softirq handler is registered at run-time via open_softirq(), which takes two parameters: the softirq's index and its handler function.

The softirq handlers run with interrupts enabled and cannot sleep.

While a handler runs, softirqs on the current processor are disabled.

Another processor, however, can execute other softirqs. If the same softirq is raised again while it is executing, another processor can run it simultaneously. This means that any shared data—even global data used only within the softirq handler—needs proper locking.

Consequently, most softirq handlers resort to per-processor data and other tricks to avoid explicit locking and provide excellent scalability.

Raising Your Softirq

To mark it pending, so it is run at the next invocation of do_softirq(), call raise_softirq(). This function disables interrupts prior to actually raising the softirq and then restores them to their previous state. If interrupts are already off, the function raise_softirq_irqoff() can be used as a small optimization. Softirqs are most often raised from within interrupt handlers.

Tasklets

Tasklets are a bottom-half mechanism built on top of softirqs. Tasklets are represented by two softirqs: HI_SOFTIRQ and TASKLET_SOFTIRQ. The only difference in these types is that the HI_SOFTIRQ-based tasklets run prior to the TASKLET_SOFTIRQ-based tasklets.

Tasklets are represented by the tasklet_struct structure. Each structure represents a unique tasklet. The structure is declared in <linux/interrupt.h>:

```
struct tasklet_struct {
   struct tasklet_struct *next;  /* next tasklet in the list */
   unsigned long state;         /* state of the tasklet */
   atomic_t count;              /* reference counter */
   void (*func)(unsigned long);  /* tasklet handler function */
   unsigned long data;          /* argument to the tasklet function */
};
```

The func member is the tasklet handler and receives data as its sole argument. The state member is exactly zero, TASKLET_STATE_SCHED, or TASKLET_STATE_RUN. TASKLET_STATE_SCHED denotes a tasklet that is scheduled to run, and TASKLET_STATE_RUN denotes a tasklet that is running.

As an optimization, TASKLET_STATE_RUN is used only on multiprocessor machines because a uniprocessor machine always knows whether the tasklet is running.

The count field is used as a reference count for the tasklet. If it is nonzero, the tasklet is disabled and cannot run; if it is zero, the tasklet is enabled and can run if marked pending.

Scheduled tasklets are stored in two per-processor structures: tasklet_vec (for regular tasklets) and tasklet_hi_vec (for high-priority tasklets). Both of these structures are linked lists of tasklet_struct structures. Each tasklet_struct structure in the list represents a different tasklet.

let’s look at the steps tasklet_schedule() undertakes:

1. Check whether the tasklet's state is TASKLET _ STATE_SCHED. If it is, the tasklet is already scheduled to run and the function can immediately return.
2. Call __tasklet_schedule().
3. Save the state of the interrupt system, and then disable local interrupts.This ensures that nothing on this processor will mess with the tasklet code while tasklet_schedule() is manipulating the tasklets.
4. Add the tasklet to be scheduled to the head of the tasklet_vec or tasklet_hi_vec linked list, which is unique to each processor in the system.
5. Raise the TASKLET_SOFTIRQ or HI_SOFTIRQ softirq, so do_softirq() executes this tasklet in the near future.
6. Restore interrupts to their previous state and return.

Because most tasklets and softirqs are marked pending in interrupt handlers, do_softirq() most likely runs when the last interrupt returns. Because TASKLET_SOFTIRQ or HI_SOFTIRQ is now raised, do_softirq() executes the associated handlers. These handlers, tasklet_action() and tasklet_hi_action(), are the heart of tasklet processing. Let's look at the steps these handlers perform:

1. Disable local interrupt delivery and retrieve the tasklet_vec or tasklet_hi_vec list for this processor.
2. Clear the list for this processor by setting it equal to NULL.
3. Enable local interrupt delivery. There is no need to restore them to their previous state because this function knows that they were always originally enabled.
4. Loop over each pending tasklet in the retrieved list.
5. If this is a multiprocessing machine, check whether the tasklet is running on another processor by checking the TASKLET_STATE_RUN flag. If it is currently running, do not execute it now and skip to the next pending tasklet.
6. If the tasklet is not currently running, set the TASKLET_STATE_RUN flag, so another processor will not run it.
7. Check for a zero-count value, to ensure that the tasklet is not disabled. If the tasklet is disabled, skip it and go to the next pending tasklet.
8. Run the tasklet handler.
9. After the tasklet runs, clear the TASKLET_STATE_RUN flag in the tasklet's state field.
10. Repeat for the next pending tasklet, until there are no more scheduled tasklets waiting to run.

You can create tasklets statically or dynamically, depending on whether you have (or want) a direct or indirect reference to the tasklet.

As with softirqs, tasklets cannot sleep. This means you cannot use semaphores or other blocking functions in a tasklet.

Tasklets also run with all interrupts enabled, so you must take precautions if your tasklet shares data with an interrupt handler. Unlike softirqs, however, two of the same tasklets never run concurrently. If your tasklet shares data with another tasklet or softirq, you need to use proper locking.

To schedule a tasklet for execution, tasklet_schedule() is called and passed a pointer to the relevant tasklet_struct:

```
tasklet_schedule(&my_tasklet); /* mark my_tasklet as pending */
```

If the same tasklet is scheduled again, before it has had a chance to run, it still runs only once. If it is already running, the tasklet is rescheduled and runs again.

You can remove a tasklet from the pending queue via tasklet_kill(). This function receives a pointer as a lone argument to the tasklet's tasklet_struct. This function first waits for the tasklet to finish executing and then it removes the tasklet from the queue. Nothing stops some other code from rescheduling the tasklet. This function must not be used from interrupt context because it sleeps.

Softirq processing is aided by a set of per-processor kernel threads. These kernel threads help in the processing of softirqs when the system is overwhelmed with softirqs.

The solution ultimately implemented in the kernel is to not immediately process reactivated softirqs. Instead, if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load. The kernel threads run with the lowest possible priority (nice value of 19), which ensures they do not run in lieu of anything important.

There is one thread per processor, each named ksoftirqd/n where n is the processor number. On a two-processor system, they are ksoftirqd/0 and ksoftirqd/1. Having a thread on each processor ensures an idle processor, if available, can always service softirqs. After the threads are initialized, they run a tight loop similar to this:

```
for (;;) {
   if (!softirq_pending(cpu))
      schedule();
   set_current_state(TASK_RUNNING);
   while (softirq_pending(cpu)) {
      do_softirq();
      if (need_resched())
         schedule();
   }
   set_current_state(TASK_INTERRUPTIBLE);
}
```

Work Queues

Work queues defer work into a kernel thread which always runs in process context. Thus, code deferred to a work queue has all the usual benefits of process context. Most important, work queues are schedulable and can therefore sleep. If you need a schedulable entity to perform your bottom-half processing, you need work queues. They are the only bottom-half mechanisms that run in process context, and thus the only ones that can sleep.

Implementing Work Queues

Work queues let your driver create a special worker thread to handle deferred work. The work queue subsystem, however, implements and provides a default worker thread for handling work. Therefore, in its most common form, a work queue is a simple interface for deferring work to a generic kernel thread.

The default worker threads are called events/n where n is the processor number.

The worker threads are represented by the workqueue_struct structure. This structure, defined in kernel/workqueue.c, contains an array of struct cpu_workqueue_struct, one per processor on the system. Because the worker threads exist on each processor in the system, there is one of these structures per worker thread, per processor, on a given machine. The cpu_workqueue_struct is the core data structure and is also defined in kernel/workqueue.c:

```
struct cpu_workqueue_struct {
   spinlock_t lock;          /* lock protecting this structure */
   struct list_head worklist;   /* list of work */
   wait_queue_head_t more_work;
   struct work_struct *current_struct;
   struct workqueue_struct *wq; /* associated workqueue_struct */
   task_t *thread;           /* associated thread */
};
```

Note that each type of worker thread has one workqueue_struct associated to it. Inside, there is one cpu_workqueue_struct for every thread and, thus, every processor, because there is one worker thread on each processor.

All worker threads are implemented as normal kernel threads running the worker_thread() function. After initial setup, this function enters an infinite loop and goes to sleep. When work is queued, the thread is awakened and processes the work. When there is no work left to process, it goes back to sleep. The work is represented by the work_struct structure, defined in <linux/workqueue.h>:

```
struct work_struct {
   atomic_long_t data;
   struct list_head entry;
   work_func_t func;
};
```

These structures are strung into a linked list, one for each type of queue on each processor.When a worker thread wakes up, it runs any work in its list. As it completes work, it removes the corresponding work_struct entries from the linked list. When the list is empty, it goes back to sleep.

The core of worker_thread is simplified as follows:

```
for (;;) {
   prepare_to_wait(&cwq->more_work, &wait, TASK_INTERRUPTIBLE);
   if (list_empty(&cwq->worklist))
      schedule();
   finish_wait(&cwq->more_work, &wait);
   run_workqueue(cwq);
}
```

This function performs the following functions, in an infinite loop:

1. The thread marks itself sleeping (the task's state is set to TASK_INTERRUPTIBLE) and adds itself to a wait queue.
2. If the linked list of work is empty, the thread calls schedule() and goes to sleep.
3. If the list is not empty, the thread does not go to sleep. Instead, it marks itself TASK_RUNNING and removes itself from the wait queue.
4. If the list is nonempty, the thread calls run_workqueue() to perform the deferred work.

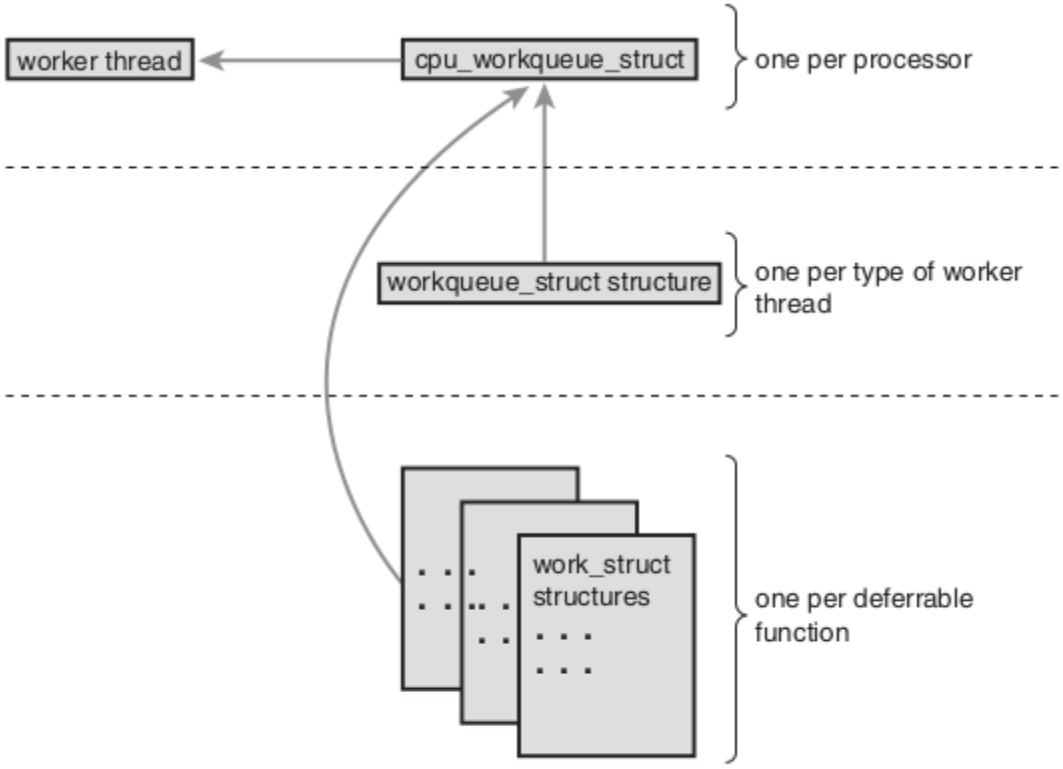

Figure 8.1 The relationship between work, work queues, and the worker threads.

At the highest level, there are worker threads. There can be multiple types of worker threads; there is one worker thread per processor of a given type. Parts of the kernel can create worker threads as needed. By default, there is the events worker thread. Each worker thread is represented by the cpu_workqueue_struct structure. The workqueue_struct structure represents all the worker threads of a given type. The driver creates work, which it wants to defer to later.
The work_struct structure represents this work. This structure contains a pointer to the function that handles the deferred work. The work is submitted to a specific worker thread. The worker thread then wakes up and performs the queued work.

Work Queue Handler

The prototype for the work queue handler is:

```
void work_handler(void *data)
```

A worker thread executes this function, so the function runs in process context. By default, interrupts are enabled and no locks are held. If needed, the function can sleep. The work is scheduled immediately and is run as soon as the events worker thread on the current processor wakes up.

Which Bottom Half Should I Use?

Softirqs, by design, provide the least serialization. This requires softirq handlers to go through extra steps to ensure that shared data is safe because two or more softirqs of the same type may run concurrently on different processors. If the code in question is already highly threaded, such as in a networking subsystem that is chest-deep in per-processor variables, softirqs make a good choice. They are certainly the fastest alternative for timing-critical and high-frequency uses.

Tasklets make more sense if the code is not finely threaded. They have a simpler interface and, because two tasklets of the same type might not run concurrently, they are easier to implement. Tasklets are effectively softirqs that do not run concurrently. A driver developer should always choose tasklets over softirqs, unless prepared to utilize per-processor variables or similar magic to ensure that the softirq can safely run concurrently on multiple processors.

If the deferred work needs to run in process context, the only choice of the three is work queues. If process context is not a **No index entries found.**requirement (specifically, if you have no need to sleep), softirqs or tasklets are perhaps better suited. Work queues involve the highest overhead because they involve kernel threads and, therefore, context switching. This doesn't mean they are inefficient, but in light of thousands of interrupts hitting per second (as the networking subsystem might experience), other methods make more sense. However, work queues are sufficient for most situations.

The following table is a comparison between the three bottom-half interfaces:

Bottom Half	Context	Inherent Serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queues	Process	None (scheduled as process context)

Locking Between the Bottom Halves

It is crucial to protect shared data from concurrent access while using bottom halves, even on a single processor machine. A bottom half can run at virtually any moment.

One benefit of tasklets is that they are serialized with respect to themselves. The same tasklet will not run concurrently, even on two different processors. This means you do not have to worry about intra-tasklet concurrency issues. Inter-tasklet concurrency (when two different tasklets share the same data) requires proper locking. Because softirqs provide no serialization, (even two instances of the same softirq might run simultaneously), all shared data needs an appropriate lock.

If process context code and a bottom half share data, you need to disable bottom-half processing and obtain a lock before accessing the data. Doing both ensures local and SMP protection and prevents a deadlock.

If interrupt context code and a bottom half share data, you need to disable interrupts and obtain a lock before accessing the data. This also ensures both local and SMP protection and prevents a deadlock.

Any shared data in a work queue requires locking, too. The locking issues are no different from normal kernel code because work queues run in process context.

Disabling Bottom Halves

To safely protect shared data, you usually need to obtain a lock and disable bottom halves. The calls can be nested; only the final call to local_bh_enable() actually enables bottom halves.

Because work queues run in process context, there are no issues with asynchronous execution. Because softirqs and tasklets can occur asynchronously, kernel code may need to disable them. With work queues, on the other hand, protecting shared data is the same as in any process context.

9)

Code paths that access and manipulate shared data are called critical regions.

Causes of concurrency in kernel:

- Interrupts: An interrupt can occur asynchronously at almost any time, interrupting the currently executing code.
- Softirqs and tasklets: The kernel can raise or schedule a softirq or tasklet at almost any time, interrupting the currently executing code.
- Kernel pre-emption: Because the kernel is pre-emptive, one task in the kernel can pre-empt another.
- Sleeping and synchronization with user-space: A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process.
- Symmetrical multiprocessing: Two or more processors can execute kernel code at exactly the same time.

Knowing What to Protect

Most global kernel data structures require locking. A good rule of thumb is that if another thread of execution can access the data, the data needs some sort of locking; if anyone else can see it, lock it. Remember to lock data, not code.

Ask yourself these questions whenever you write kernel code:

- Is the data global? Can a thread of execution other than the current one access it?
- Is the data shared between process context and interrupt context? Is it shared between two different interrupt handlers?
- If a process is pre-empted while accessing this data, can the newly scheduled process access the same data?
- Can the current process sleep (block) on anything? If it does, in what state does that leave any shared data?
- What prevents the data from being freed out from under me?
- What happens if this function is called again on another processor?
- Given the proceeding points, how am I going to ensure that my code is safe from concurrency?

A deadlock is a condition involving one or more threads of execution and one or more resources, such that each thread waits for one of the resources, but all the resources are already held. The threads all wait for each other, but they never make any progress toward releasing the resources that they already hold. Therefore, none of the threads can continue. The simplest example of a deadlock is the self-deadlock. If a thread of execution attempts to acquire a lock it already holds, it has to wait

for the lock to be released. But it will never release the lock, because it is busy waiting for the lock, and the result is deadlock:

acquire lock

acquire lock, again

wait for lock to become available

Similarly, consider n threads and n locks. If each thread holds a lock that the other thread wants, all threads block while waiting for their respective locks to become available. The most common example is with two threads and two locks, which is often called the deadly embrace or the ABBA deadlock:

Thread 1	**Thread 2**
acquire lock A	acquire lock B
try to acquire lock B	try to acquire lock A
wait for lock B	wait for lock A

you can write deadlock-free code following the rules below:

- Implement lock ordering. Nested locks must always be obtained in the same order.
- Prevent starvation.
- Do not double acquire the same lock.

The first point is most important and worth stressing. If two or more locks are acquired at the same time, they must always be acquired in the same order.

Contention and Scalability

The term lock contention, or simply contention, describes a lock currently in use but that another thread is trying to acquire. A lock that is highly contended often has threads waiting to acquire it.

Scalability is a measurement of how well a system can be expanded. The granularity of locking is a description of the size or amount of data that a lock protects.

10)

atomic operations provide instructions that execute atomically, without interruption. The kernel provides two sets of interfaces for atomic operations: one that operates on integers and another that operates on individual bits.

The atomic integer methods operate on a special data type, atomic_t. This special type is used, as opposed to having the functions work directly on the C int type, for several reasons:

1. Having the atomic functions accept only the atomic_t type ensures that the atomic operations are used only with these special types. Likewise, it also ensures that the data types are not passed to any non-atomic functions.

2. The use of atomic_t ensures the compiler does not optimize access to the value, it is important the atomic operations receive the correct memory address and not an alias.
3. Use of atomic_t can hide any architecture-specific differences in its implementation.

The atomic_t type is defined in <linux/types.h>:

```
typedef struct {
   volatile int counter;
} atomic_t;
```

The atomic operations are typically implemented as inline functions with inline assembly. In the case where a specific function is inherently atomic, the given function is usually just a macro.

atomic_t is 32-bit even on 64-bit architectures. Instead, the atomic64_t type provides a 64-bit atomic integer. As with atomic_t, the atomic64_t type is just a simple wrapper around an integer type a long:

```
typedef struct {
   volatile long counter;
} atomic64_t;
```

Atomic Bitwise Operations

The functions that operate at the bit level are architecture-specific and defined in <asm/bitops.h>. The bitwise functions operate on generic memory addresses. The arguments are a pointer and a bit number. Bit zero is the least significant bit of the given address.

Spin Locks

A spin lock is a lock that can be held by at most one thread of execution. The fact that a contended spin lock causes threads to spin while waiting for the lock to become available is salient. It is not wise to hold a spin lock for a long time. So, spin lock is a lightweight single-holder lock that should be held for short durations.

Only one thread is allowed in the critical region at a time. This provides the needed protection from concurrency on multiprocessing machines. On uniprocessor machines, the locks compile away and do not exist; they simply act as markers to disable and enable kernel pre-emption. If kernel pre-empt is turned off, the locks compile away entirely.

Note that the Linux kernel's spin locks are not recursive.

If a lock is used in an interrupt handler, you must also disable local interrupts before obtaining the lock. Otherwise, it is possible for a double-acquire deadlock. Note that you need to disable interrupts only on the current processor.

The kernel provides an interface that conveniently disables interrupts and acquires the lock. The routine spin_lock_irqsave() saves the current state of interrupts, disables them locally, and then obtains the given lock. Conversely, spin_unlock_irqrestore() unlocks the given lock and returns interrupts to their previous state. Note that the flags variable is seemingly passed by value. This is because the lock routines are implemented partially as macros.

Spin Locks and Bottom Halves

certain locking precautions must be taken when working with bottom halves. Because a bottom half might pre-empt process context code, if data is shared between a bottom-half process context, you must protect the data in process context with both a lock and the disabling of bottom halves. Likewise, because an interrupt handler might pre-empt a bottom half, if data is shared between an interrupt handler and a bottom half, you must both obtain the appropriate lock and disable interrupts.

Recall that two tasklets of the same type do not ever run simultaneously. Thus, there is no need to protect data used only within a single type of tasklet. If the data is shared between two different tasklets, you must obtain a normal spin lock before accessing the data in the bottom half. You do not need to disable bottom halves because a tasklet never pre-empts another running tasklet on the same processor.

With softirqs, regardless of whether it is the same softirq type, if data is shared by softirqs, it must be protected with a lock. Recall that softirqs, even two of the same type, might run simultaneously on multiple processors in the system. A softirq never pre-empts another softirq running on the same processor, however, so disabling bottom halves is not needed.

Reader-Writer Spin Locks

Linux kernel provides reader-writer spin locks. Reader-writer spin locks provide separate reader and writer variants of the lock. One or more readers can concurrently hold the reader lock. The writer lock, conversely, can be held by at most one writer with no concurrent readers.

Normally, the readers and writers are in entirely separate code paths. Note that you cannot "upgrade" a read lock to a write lock. For example, executing these two functions as shown will deadlock:

read_lock(&mr_rwlock);

write_lock(&mr_rwlock);

See Table for a full listing of the reader-writer spin lock methods:

Method	Description
read_lock()	Acquires given lock for reading
read_lock_irq()	Disables local interrupts and acquires given lock for reading
read_lock_irqsave()	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
read_unlock()	Releases given lock for reading
read_unlock_irq()	Releases given lock and enables local interrupts

Method	Description
read_unlock_irqrestore()	Releases given lock and restores local interrupts to the given previous state
write_lock()	Acquires given lock for writing
write_lock_irq()	Disables local interrupts and acquires the given lock for writing
write_lock_irqsave()	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing
write_unlock()	Releases given lock
write_unlock_irq()	Releases given lock and enables local interrupts
write_unlock_irqrestore()	Releases given lock and restores local interrupts to given previous state
write_trylock()	Tries to acquire given lock for writing; if unavailable, returns nonzero
rwlock_init()	Initializes given rwlock_t

A final important consideration in using the Linux reader-writer spin locks is that they favour readers over writers.

Semaphores

Semaphores in Linux are sleeping locks. When a task attempts to acquire a semaphore that is unavailable, the semaphore places the task onto a wait queue and puts the task to sleep. The processor is then free to execute other code. When the semaphore becomes available, one of the tasks on the wait queue is awakened so that it can then acquire the semaphore.

The uses of semaphores versus spin locks.:

- Because the contending tasks sleep while waiting for the lock to become available, semaphores are well suited to locks that are held for a long time.
- Conversely, semaphores are not optimal for locks that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.
- Because a thread of execution sleeps on lock contention, semaphores must be obtained only in process context because interrupt context is not schedulable.
- You can sleep while holding a semaphore because you will not deadlock when another process acquires the same semaphore.

- You cannot hold a spin lock while you acquire a semaphore, because you might have to sleep while waiting for the semaphore, and you cannot sleep while holding a spin lock.

Counting and Binary Semaphores

A feature of semaphores is that they can allow for an arbitrary number of simultaneous lock holders. Whereas spin locks permit at most one task to hold the lock at a time, the number of permissible simultaneous holders of semaphores can be set at declaration time. This value is called the usage count or simply the count.

- binary semaphore: the count is equal to one.
- counting semaphore: the count initialized to a nonzero value greater than one.
- up: used to release a semaphore upon completion of a critical region.
- down: used to acquire a semaphore by decrementing the count by one.

to initialize a dynamically created semaphore to which you have only an indirect pointer reference, just call sema_init(), where sem is a pointer and count is the usage count of the semaphore:

```
sema_init(sem, count);
```

Similarly, to initialize a dynamically created mutex:

```
init_MUTEX(sem);
```

Using Semaphores

Method	Description
sema_init(struct semaphore *, int)	Initializes the dynamically created semaphore to the given count
init_MUTEX(struct semaphore *)	Initializes the dynamically created semaphore with a count of one
init_MUTEX_LOCKED(struct semaphore *)	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
down_interruptible (struct semaphore *)	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
down(struct semaphore *)	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
down_trylock(struct semaphore *)	Tries to acquire the given semaphore and immediately return nonzero if it is contended
up(struct semaphore *)	Releases the given semaphore and wakes a waiting task, if any

Reader-Writer Semaphores

Reader-writer semaphores are represented by the struct rw_semaphore type, which is declared in <linux/rwsem.h>. All reader-writer semaphores's usage count is one. All reader-writer locks use uninterruptible sleep, so there is only one version of each down():

```
static DECLARE_RWSEM(mr_rwsem);

/* attempt to acquire the semaphore for reading ... */

down_read(&mr_rwsem);

/* critical region (read only) ... */

/* release the semaphore */

up_read(&mr_rwsem);

/* ... */

/* attempt to acquire the semaphore for writing ... */

down_write(&mr_rwsem);

/* critical region (read and write) ... */

/* release the semaphore */

up_write(&mr_sem);
```

Reader-writer semaphores have a unique method that their reader-writer spin lock cousins do not have: downgrade_write(). This function atomically converts an acquired write lock to a read lock.

Mutexes

The term "mutex" is a generic name to refer to any sleeping lock that enforces mutual exclusion. But now, "mutex" also a specific type of sleeping lock that implements mutual exclusion. The mutex has a stricter, narrower use case:

- Only one task can hold the mutex at a time. That is, the usage count on a mutex is always one.
- Whoever locked a mutex must unlock it. Most use cases, cleanly lock and unlock from the same context.
- Recursive locks and unlocks are not allowed.
- A process cannot exit while holding a mutex.
- A mutex cannot be acquired by an interrupt handler or bottom half, even with mutex_trylock().
- A mutex can be managed only via the official API.

Semaphores Versus Mutexes

Unless one of mutex’s additional con- strains prevent you from using them, prefer the new mutex type to semaphores.

Spin Locks Versus Mutexes

Only a spin lock can be used in interrupt context, whereas only a mutex can be held while a task sleeps.

Requirement	Recommended Lock
Low overhead locking	Spin lock is preferred
Short lock hold time	Spin lock is preferred
Long lock hold time	Mutex is preferred
Need to lock from interrupt context	Spin lock is required
Need to sleep while holding lock	Mutex is required

Completion Variables

Using completion variables is an easy way to synchronize between two tasks in the kernel when one task needs to signal to the other that an event has occurred.

Completion variables are represented by the struct completion type, which is defined in <linux/completion.h>. A common usage is to have a completion variable dynamically created as a member of a data structure. Kernel code waiting for the initialization of the data structure calls wait_for_completion(). When the initialization is complete, the waiting tasks are awakened via a call to completion().

Sequential Locks

The sequential lock provides a simple mechanism for reading and writing shared data. It works by maintaining a sequence counter. Whenever the data in question is written to, a lock is obtained and a sequence number is incremented. Prior to and after reading the data, the sequence number is read. If the values are the same, a write did not begin in the middle of the read. Further, if the values are even, a write is not underway.

The write path is:

```
write_seqlock(&mr_seq_lock);

/* write lock is obtained... */

write_sequnlock(&mr_seq_lock);
```

This looks like normal spin lock code. The oddness comes in with the read path, which is quite a bit different:

```
unsigned long seq;

do {

    seq = read_seqbegin(&mr_seq_lock);
```

```
    /* read data here ... */
} while (read_seqretry(&mr_seq_lock, seq));
```

Seq locks favour writers over readers. Seq locks are ideal when your locking needs meet most or all these requirements:

- Your data has a lot of readers.
- Your data has few writers.
- Although few in number, you want to favour writers over readers and never allow readers to starve writers.
- Your data is simple.

Pre-emption Disabling

the kernel pre-emption code uses spin locks as markers of nonpreemptive regions. If a spin lock is held, the kernel is not pre-emptive.

Sometimes we do not require a spin lock, instead, preempt_disable() can disables kernel pre-emption. The call is nestable; you can call it any number of times. For each call, a corresponding call to preempt_enable() is required. The final corresponding call to preempt_enable() reenables pre-emption.

The pre-emption count stores the number of held locks and preempt_disable() calls. If the number is zero, the kernel is pre-emptive. If the value is one or greater, the kernel is not preemptive.

A listing of kernel pre-emption-related functions:

Function	Description
preempt_disable()	Disables kernel pre-emption by incrementing the pre-emption counter
preempt_enable()	Decrements the pre-emption counter and checks and services any pending reschedules if the count is now zero
preempt_enable_no_resched()	Enables kernel pre-emption but does not check for any pending reschedules
preempt_count()	Returns the pre-emption count

Ordering and Barriers

When dealing with synchronization between multiple processors or with hardware devices, it is sometimes a requirement that memory-reads (loads) and memory-writes (stores) issue in the order specified in your program code. A full listing of the memory and compiler barrier methods provided by all architectures in the Linux kernel:

Barrier	Description
rmb()	Prevents loads from being reordered across the barrier
read_barrier_depends()	Prevents data-dependent loads from being reordered across the barrier
wmb()	Prevents stores from being reordered across the barrier
mb()	Prevents load or stores from being reordered across the barrier
smp_rmb()	Provides an rmb() on SMP, and on UP provides a barrier()
smp_read_barrier_depends()	Provides a read_barrier_depends() on SMP, and provides a barrier() on UP
smp_wmb()	Provides a wmb() on SMP, and provides a barrier() on UP
smp_mb()	Provides an mb() on SMP, and provides a barrier() on UP
barrier()	Prevents the compiler from optimizing stores or loads across the barrier

11)

The hardware provides a system timer that the kernel uses to gauge the passing of time. When the system timer goes off, it issues an interrupt that the kernel handles via a special interrupt handler. The period is called a tick and is equal to 1/(tick rate) seconds.

Some of the work executed periodically by the timer interrupt:

- Updating the system uptime
- Updating the time of day
- On an SMP system, ensuring that the scheduler runqueues are balanced and, if not, balancing them
- Running any dynamic timers that have expired
- Updating resource usage and processor time statistics

The Tick Rate: HZ

The frequency of the system timer is programmed on system boot based on a static preprocessor define, HZ. The kernel defines the value in <asm/param.h>. The tick rate has a frequency of HZ hertz and a period of 1/HZ seconds.

The Ideal HZ Value

Advantages with a Larger HZ

- Kernel timers execute with finer resolution and increased accuracy.
- System calls such as poll() and select() that optionally employ a timeout value execute with improved precision.
- Measurements, such as resource usage or the system uptime, are recorded with a finer resolution.
- Process pre-emption occurs more accurately.

Disadvantages with a Larger HZ

A higher tick rate implies more frequent timer interrupts, which implies higher overhead, because the processor must spend more time executing the timer interrupt handler.

Jiffies

The global variable jiffies hold the number of ticks that have occurred since the system booted. The kernel initializes jiffies to a special initial value, causing the variable to overflow more often, catching bugs. When the actual value of jiffies is sought, this “offset” is first subtracted.

Internal Representation of Jiffies

The jiffies variable has always been an unsigned long, and therefore 32 bits in size on 32-bit architectures and 64-bits on 64-bit architectures.

jiffies is defined as an unsigned long in <linux/jiffies.h>:

```
extern unsigned long volatile jiffies;

extern u64 jiffies_64;
```

The ld(1) script used to link the main kernel image then overlays the jiffies variable over the start of the jiffies_64 variable:

```
jiffies = jiffies_64;
```

Thus, jiffies is the lower 32 bits of the full 64-bit jiffies_64 variable. Because most code uses jiffies simply to measure elapses in time, most code cares about only the lower 32 bits. The time management code uses the entire 64 bits, however, and thus prevents overflow of the full 64-bit value.

Jiffies Wraparound

The jiffies variable, experiences overflow when its value is increased beyond its maximum storage limit. When the tick count is equal to this maximum and it is incremented, it wraps around to zero.

The kernel provides four macros for comparing tick counts that correctly handle wraparound in the tick count. They are in <linux/jiffies.h>. Listed here are simplified versions of the macros:

```
#define time_after(unknown, known) ((long)(known) - (long)(unknown) < 0)

#define time_before(unknown, known) ((long)(unknown) - (long)(known) < 0)

#define time_after_eq(unknown, known) ((long)(unknown) - (long)(known) >= 0)
```

```
#define time_before_eq(unknown, known) ((long)(known) - (long)(unknown) >= 0)
```

The unknown parameter is typically jiffies and the known parameter is the value against which you want to compare.

User-Space and HZ

The kernel defined USER_HZ, which is the HZ value that user-space expects. The function jiffies_to_clock_t(), defined in kernel/time.c, is then used to scale a tick count in terms of HZ to a tick count in terms of USER_HZ. The function jiffies_64_to_clock_t() is provided to convert a 64-bit jiffies value from HZ to USER_HZ units.

Hardware Clocks and Timers

Real-Time Clock

The real-time clock (RTC) provides a non-volatile device for storing the system time. Its primary importance is only during boot, when the xtime variable is initialized.

System Timer

The system timer is to provide a mechanism for driving an interrupt at a periodic rate. On x86, the primary system timer is the programmable interrupt timer (PIT).

The Timer Interrupt Handler

The timer interrupt is broken into two pieces: an architecture-dependent and an architecture-independent routine.

The architecture-dependent routine is registered as the interrupt handler for the system timer and, thus, runs when the timer interrupt hits.

Most handlers perform at least the following work:

- Obtain the xtime_lock lock, which protects access to jiffies_64 and the wall time value, xtime.
- Acknowledge or reset the system timer as required.
- Periodically save the updated wall time to the real time clock.
- Call the architecture-independent timer routine, tick_periodic().

The architecture-independent routine, tick_periodic(), performs much more work:

- Increment the jiffies_64 count by one.
- Update resource usages, such as consumed system and user time, for the currently running process.
- Run any dynamic timers that have expired.
- Execute scheduler_tick().
- Update the wall time, which is stored in xtime.
- Calculate the infamous load average.

```
static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {
        write_seqlock(&xtime_lock);

        /* Keep track of the next tick event */
        tick_next_period = ktime_add(tick_next_period, tick_period);

        do_timer(1);
        write_sequnlock(&xtime_lock);
    }

    update_process_times(user_mode(get_irq_regs()));
    profile_tick(CPU_PROFILING);
}
```

do_timer() is responsible for actually performing the increment to jiffies_64:

```
void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks;
    update_wall_time();
    calc_global_load();
}
```

The function update_wall_time() updates the wall time in accordance with the elapsed ticks, whereas calc_global_load() updates the system's load average statistics.

When do_timer() ultimately returns, update_process_times() is invoked to update various statistics that a tick has elapsed, noting via user_tick whether it occurred in user-space or kernel-space:

```
    void update_process_times(int user_tick)
    {
        struct task_struct *p = current;
        int cpu = smp_processor_id();

        /* Note: this timer irq context must be accounted for as well. */
```

```
    account_process_tick(p, user_tick);
    run_local_timers();
    rcu_check_callbacks(cpu, user_tick);
    printk_tick();
    scheduler_tick();
    run_posix_cpu_timers(p);
}
```

The account_process_tick() function does the actual updating of the process's times:

```
void account_process_tick(struct task_struct *p, int user_tick)
{
    cputime_t one_jiffy_scaled = cputime_to_scaled(cputime_one_jiffy);
    struct rq *rq = this_rq();

    if (user_tick)
        account_user_time(p, cputime_one_jiffy, one_jiffy_scaled);
    else if ((p != rq->idle) || (irq_count() != HARDIRQ_OFFSET))
        account_system_time(p, HARDIRQ_OFFSET, cputime_one_jiffy,
                one_jiffy_scaled);
    else
        account_idle_time(cputime_one_jiffy);
}
```

Next, the run_local_timers() function marks a softirq to handle the execution of any expired timers.

Finally, the scheduler_tick() function decrements the currently running process's time slice and sets need_resched if needed.

The Time of Day

The current time of day (the wall time) is defined in kernel/time/timekeeping.c:

```
struct timespec xtime;
```

The timespec data structure is defined in <linux/time.h> as:

```
struct timespec {
    __kernel_time_t tv_sec;   /* seconds */
    long tv_nsec;             /* nanoseconds */
```

```
};
```

Reading or writing the xtime variable requires the xtime_lock lock, which is not a normal spinlock but a seqlock.

```
write_seqlock(&xtime_lock);

/* update xtime ... */

write_sequnlock(&xtime_lock);
```

Reading xtime requires the use of the read_seqbegin() and read_seqretry() functions:

```
unsigned long seq;

do {

    unsigned long lost;

    seq = read_seqbegin(&xtime_lock);


    usec = timer->get_offset();

    lost = jiffies - wall_jiffies;

    if (lost)

        usec += lost * (1000000 / HZ);

    sec = xtime.tv_sec;

    usec += (xtime.tv_nsec / 1000);

} while (read_seqretry(&xtime_lock, seq));
```

This loop repeats until the reader is assured that it read the data without an intervening write. If the timer interrupt occurred and updated xtime during the loop, the returned sequence number is invalid and the loop repeats.

The primary user-space interface for retrieving the wall time is gettimeofday(), which is implemented as sys_gettimeofday() in kernel/time.c:

```
asmlinkage long sys_gettimeofday(struct timeval *tv, struct timezone *tz)

{

    if (likely(tv)) {

        struct timeval ktv;

        do_gettimeofday(&ktv);

        if (copy_to_user(tv, &ktv, sizeof(ktv)))

            return -EFAULT;

    }
```

```
        if (unlikely(tz)) {
            if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
            return -EFAULT;
        }
        return 0;
    }
```

Timers

Timers is a tool for delaying work a specified amount of time.

Using Timers

Timers are represented by struct timer_list, which is defined in <linux/timer.h>:

```
    struct timer_list {
        struct list_head entry;         /* entry in linked list of timers */
        unsigned long expires;          /* expiration value, in jiffies */
        void (*function)(unsigned long);  /* the timer handler function */
        unsigned long data;             /* lone argument to the handler */
        struct tvec_t_base_s *base;     /* internal timer field, do not touch */
    };
```

The kernel provides a family of timer-related interfaces to make timer management easy. Everything is declared in <linux/timer.h>. Most of the actual implementation is in kernel/timer.c.

The first step in creating a timer is defining it:

```
    struct timer_list my_timer;
```

Next, the timer's internal values must be initialized. This is done via a helper function and must be done prior to calling any timer management functions on the timer:

```
    init_timer(&my_timer);
```

Now you fill out the remaining values as required:

```
    my_timer.expires = jiffies + delay;   /* timer expires in delay ticks */
    my_timer.data = 0;                    /* zero is passed to the timer handler */
    my_timer.function = my_function;      /* function to run when timer expires */
```

As you can see from the timer_list definition, the function must match this prototype:

```
    void my_timer_function(unsigned long data);
```

The data parameter enables you to register multiple timers with the same handler, and differentiate between them via the argument. If you do not need the argument, you can simply pass zero.

Finally, you activate the timer:

```
add_timer(&my_timer);
```

Typically, timers are run fairly close to their expiration; however, they might be delayed until the first timer tick after their expiration. Consequently, timers cannot be used to implement any sort of hard real-time processing.

The kernel implements a function, mod_timer(), which changes the expiration of a given timer:

```
mod_timer(&my_timer, jiffies + new_delay);    /* new expiration */
```

The mod_timer() function can operate on timers that are initialized but not active, too. If the timer is inactive, mod_timer() activates it. The function returns zero if the timer were inactive and one if the timer were active. In either case, upon return from mod_timer(), the timer is activated and set to the new expiration.

If you need to deactivate a timer prior to its expiration, use the del_timer() function:

```
del_timer(&my_timer);
```

The function works on both active and inactive timers. If the timer is already inactive, the function returns zero; otherwise, the function returns one. Note that you do not need to call this for timers that have expired because they are automatically deactivated.

A potential race condition that must be guarded against exists when deleting timers.
When del_timer() returns, it guarantees only that the timer is no longer active. On a multiprocessing machine, however, the timer handler might already be executing on another processor. To deactivate the timer and wait until a potentially executing handler for the timer exits, use del_timer_sync():

```
del_timer_sync(&my_timer);
```

Unlike del_timer(), del_timer_sync() cannot be used from interrupt context. In almost all cases, you should use del_timer_sync() over del_timer().

Timer Implementation

After the timer interrupt, the timer interrupt handler runs update_process_times(), which calls run_local_timers():

```
void run_local_timers(void)
{
    hrtimer_run_queues();
    raise_softirq(TIMER_SOFTIRQ);   /* raise the timer softirq */
    softlockup_tick();
}
```

The TIMER_SOFTIRQ softirq is handled by run_timer_softirq(). This function runs all the expired timers on the current processor.

The kernel partitions timers into five groups based on their expiration value for efficiencies.

Delaying Execution

Busy Looping

Busy looping only used when the time you want to delay is some integer multiple of the tick rate on precision is not important.

```
unsigned long timeout = jiffies + 10;     /* ten ticks */

while (time_before(jiffies, timeout))

    ;
```

The loop continues until jiffies is larger than delay, which occurs only after 10 clock ticks have passed.

A better solution would be to reschedule your process to allow the processor to accomplish other work while your code waits:

```
unsigned long delay = jiffies + 5*HZ;

while (time_before(jiffies, delay))

    cond_resched();
```

The call to cond_resched() schedules a new process, but only if need_resched is set. Note that because this approach invokes the scheduler, you can only make use of it from process context.

Small Delays

The kernel provides three functions for microsecond, nanosecond, and millisecond delays, defined in <linux/delay.h> and <asm/delay.h>, which do not use jiffies:

```
void udelay(unsigned long usecs)

void ndelay(unsigned long nsecs)

void mdelay(unsigned long msecs)
```

schedule_timeout()

A more optimal method of delaying execution is to use schedule_timeout(). This call puts your task to sleep until at least the specified time has elapsed.

```
/* set task's state to interruptible sleep */

set_current_state(TASK_INTERRUPTIBLE);

/* take a nap and wake up in "s" seconds */

schedule_timeout(s * HZ);
```

If the code does not want to process signals, you can use TASK_UNINTERRUPTIBLE instead. The task must be in one of these two states before schedule_timeout() is called or else the task will not go to sleep.

schedule_timeout() Implementation

```
signed long schedule_timeout(signed long timeout)

{
```

```
        timer_t timer;
        unsigned long expire;
        switch (timeout)
        {
          case MAX_SCHEDULE_TIMEOUT:
                schedule();
                goto out;
          default:
                if (timeout < 0)
                {
                      printk(KERN_ERR “schedule_timeout: wrong timeout “
                         “value %lx from %p\n”, timeout,
                         __builtin_return_address(0));
                      current->state = TASK_RUNNING;
                      goto out;
                }
        }
        expire = timeout + jiffies;
        init_timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;
        add_timer(&timer);
        schedule();
        del_timer_sync(&timer);
        timeout = expire - jiffies;
    out:
        return timeout < 0 ? 0 : timeout;
    }
```

When the timer expires, it runs process_timeout():

```
    void process_timeout(unsigned long data)
```

```
{
        wake_up_process((task_t *) data);
}
```

This function puts the task in the TASK_RUNNING state and places it back on the runqueue.

Sleeping on a Wait Queue, with a Timeout

Sometimes it is desirable to wait for a specific event or wait for a specified time to elapse. In those cases, code might simply call schedule_timeout() instead of schedule() after placing itself on a wait queue.

12)

Pages

The kernel treats physical pages as the basic unit of memory management. Most 32-bit architectures have 4KB pages, whereas most 64-bit architectures have 8KB pages.

The kernel represents every physical page on the system with a struct page structure. This structure is defined in <linux/mm_types.h>. The following is a simplified the definition:

```
struct page {
   unsigned long flags;
   atomic_t _count;
   atomic_t _mapcount;
   unsigned long private;
   struct address_space *mapping;
   pgoff_t index;
   struct list_head lru;
   void *virtual;
};
```

The flags field stores the status of the page. Bit flags represent the various values, so at least 32 different flags are simultaneously available. The flag values are defined in <linux/page-flags.h>.

The _count field stores the usage count of the page. When this count reaches negative one, no one is using the page, and it becomes available for use in a new allocation. Kernel check this field with page_count(), it returns zero to indicate free and a positive nonzero integer when the page is in use.

The virtual field is the page's virtual address.

The important point to understand is that the page structure is associated with physical pages, not virtual pages. The data structure's goal is to describe physical memory, not the data contained therein.

An instance of this structure is allocated for each physical page in the system.

Zones

The kernel divides pages into different zones and use the zones to group pages of similar properties.

Linux has to deal with two shortcomings of hardware with respect to memory addressing:

- Some hardware devices can perform DMA (direct memory access) to only certain memory addresses.
- Some architectures can physically address larger amounts of memory than they can virtually address. Consequently, some memory is not permanently mapped into the kernel address space.

Due to these constraints, Linux has four primary memory zones defined in <linux/mmzone.h>:

- ZONE_DMA: This zone contains pages that can undergo DMA.
- ZONE_DMA32: Like ZOME_DMA, this zone contains pages that can undergo DMA. Unlike ZONE_DMA, these pages are accessible only by 32-bit devices. On some architectures, this zone is a larger subset of memory.
- ZONE_NORMAL: This zone contains normal, regularly mapped, pages.
- ZONE_HIGHMEM: This zone contains "high memory", which are pages not permanently mapped into the kernel's address space.

The layout of the memory zones is architecture-dependent. The following table is a listing of each zone and its consumed pages on x86-32:

Zone	Description	Physical Memory
ZONE_DMA	DMA-able pages	< 16MB
ZONE_NORMAL	Normally addressable pages	16–896MB
ZONE_HIGHMEM	Dynamically mapped pages	> 896MB

But Intel's x86-64 can fully map and handle 64-bits of memory. Thus, x86-64 has no ZONE_HIGHMEM and all physical memory is contained within ZONE_DMA and ZONE_NORMAL.

Each zone is represented by struct zone, which is defined in <linux/mmzone.h>:

```
struct zone {
    unsigned long watermark[NR_WMARK];
    unsigned long lowmem_reserve[MAX_NR_ZONES];
    struct per_cpu_pageset pageset[NR_CPUS];
    spinlock_t lock;
    struct free_area free_area[MAX_ORDER]
```

```
    spinlock_t lru_lock;
    struct zone_lru {
       struct list_head list;
       unsigned long nr_saved_scan;
    } lru[NR_LRU_LISTS];
    struct zone_reclaim_stat reclaim_stat;
    unsigned long pages_scanned;
    unsigned long flags;
    atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];
    int prev_priority;
    unsigned int inactive_ratio;
    wait_queue_head_t *wait_table;
    unsigned long wait_table_hash_nr_entries;
    unsigned long wait_table_bits;
    struct pglist_data *zone_pgdat;
    unsigned long zone_start_pfn;
    unsigned long spanned_pages;
    unsigned long present_pages;
    const char *name;
};
```

The lock field is a spin lock that protects the structure from concurrent access. It protects just the structure and not all the pages that reside in the zone. A specific lock does not protect individual pages.

The watermark array holds the minimum, low, and high watermarks for this zone. The kernel uses watermarks to set benchmarks for suitable per-zone memory consumption.

The name field is a NULL-terminated string representing the name of this zone. The kernel initializes this value during boot in mm/page_alloc.c, and the three zones are given the names DMA, Normal, and HighMem.

Getting Pages

The kernel provides one low-level mechanism for requesting memory, along with several interfaces to access it. All these interfaces allocate memory with page-sized granularity and are declared in <linux/gfp.h>. The core function is:

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

This allocates 2^{order} (1 << order) contiguous physical pages and returns a pointer to the first page's page structure; on error it returns NULL.

You can convert a given page to its logical address with the function:

```
void * page_address(struct page *page)
```

This returns a pointer to the logical address where the given physical page currently resides.

If you have no need for the actual struct page, you can call:

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
```

This function works the same as alloc_pages(), except that it directly returns the logical address of the first requested page. Because the pages are contiguous, the other pages simply follow from the first.

If you need only one page, two functions are implemented as wrappers:

```
struct page * alloc_page(gfp_t gfp_mask)
```

```
unsigned long __get_free_page(gfp_t gfp_mask)
```

These functions work the same but pass zero for the order (2^0 = one page).

Getting Zeroed Pages

If you need the returned page filled with zeros, use the function:

```
unsigned long get_zeroed_page(unsigned int gfp_mask)
```

This function works the same as __get_free_page(), except that the allocated page is then zero-filled (every bit of every byte is unset).

A listing of all the low-level page allocation methods:

Flag	Description
alloc_page(gfp_mask)	Allocates a single page and returns a pointer to its first page's page structure
alloc_pages(gfp_mask, order)	Allocates 2^{order} pages and returns a pointer to the first page's page structure
__get_free_page(gfp_mask)	Allocates a single page and returns a pointer to its logical address
__get_free_pages(gfp_mask, order)	Allocates 2^{order} pages and returns a pointer to the first page's logical address
get_zeroed_page(gfp_mask)	Allocates a single page, zero its contents and returns a pointer to its logical address

Freeing Pages

A family of functions enables you to free allocated pages when you no longer need them:

```
void __free_pages(struct page *page, unsigned int order)

void free_pages(unsigned long addr, unsigned int order)

void free_page(unsigned long addr)
```

Be careful to free only pages you allocate. Passing the wrong struct page or address, or the incorrect order, can result in corruption.

kmalloc()

The kmalloc() function obtains kernel memory in byte-sized chunks. The function declared in <linux/slab.h>:

```
void * kmalloc(size_t size, gfp_t flags)
```

The function returns a pointer to a region of memory that is at least size bytes in length. The region of memory allocated is physically contiguous. On error, it returns NULL.

gfp_mask Flags

Flags are represented by the gfp_t type, which is defined in <linux/types.h> as an unsigned int. gfp stands for __get_free_pages().

The flags are broken up into three categories:

- action modifiers: specify how the kernel is supposed to allocate the requested memory.
- zone modifiers: specify from where to allocate memory.
- type: specify a combination of action and zone modifiers for a certain type of memory allocation.

All the flags are declared in <linux/gfp.h>. The file <linux/slab.h> includes this header.

Action Modifiers

Flag	Description
__GFP_WAIT	The allocator can sleep.
__GFP_HIGH	The allocator can access emergency pools.
__GFP_IO	The allocator can start disk I/O.
__GFP_FS	The allocator can start filesystem I/O.
__GFP_COLD	The allocator should use cache cold pages.
__GFP_NOWARN	The allocator does not print failure warnings.

Flag	Description
__GFP_REPEAT	The allocator repeats the allocation if it fails, but the allocation can potentially fail.
__GFP_NOFAIL	The allocator indefinitely repeats the allocation. The allocation cannot fail.
__GFP_NORETRY	The allocator never retries if the allocation fails.
__GFP_NOMEMALLOC	The allocator does not fall back on reserves.
__GFP_HARDWALL	The allocator enforces "hardwall" cpuset boundaries.
__GFP_RECLAIMABLE	The allocator marks the pages reclaimable.
__GFP_COMP	The allocator adds compound.

Zone Modifiers

There are only three zone modifiers because there are only three zones other than ZONE_NORMAL, as in the following table:

Flag	Description
__GFP_DMA	Allocates only from ZONE_DMA
__GFP_DMA32	Allocates only from ZONE_DMA32
__GFP_HIGHMEM	Allocates from ZONE_HIGHMEM or ZONE_NORMAL

Type Flags

The table below is a list of the type flags:

Flag	Description
GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spinlock, and in other situations where you cannot sleep.
GFP_NOWAIT	Like GFP_ATOMIC, except that the call will not fall back on emergency memory pools. This increases the likelihood of the memory allocation failing.

Flag	Description
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O, which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but it will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to do to obtain the memory requested by the caller. This flag should be your default choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the preceding flags.

The following table shows which modifiers are associated with each type flag:

Flag	Modifier Flags
GFP_ATOMIC	__GFP_HIGH
GFP_NOWAIT	0
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT | __GFP_IO)
GFP_KERNEL	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_USER	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_HIGHUSER	(__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

Below is a list of the common situations and the flags to use.

Situation	Solution
Process context, can sleep	Use GFP_KERNEL.
Process context, cannot sleep	Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep.
Interrupt handler	Use GFP_ATOMIC.
Softirq	Use GFP_ATOMIC.
Tasklet	Use GFP_ATOMIC.
Need DMA-able memory, can sleep	Use (GFP_DMA | GFP_KERNEL).
Need DMA-able memory, cannot sleep	Use (GFP_DMA | GFP_ATOMIC), or perform your allocation at an earlier point when you can sleep.

kfree()

The counterpart to kmalloc() is kfree(), declared in <linux/slab.h>:

```
void kfree(const void *ptr)
```

The kfree() method frees a block of memory previously allocated with kmalloc().

vmalloc()

The vmalloc() function works in a similar fashion to kmalloc(), except vmalloc() allocates memory that is only virtually contiguous and not necessarily physically contiguous. This is similar to user-space malloc().

Most kernel code uses knalloc() and not vmalloc() to obtain memory for performance. vmalloc() is used only when absolutely necessary to obtain large regions of memory.

The vmalloc() function is declared in <linux/vmalloc.h> and defined in mm/vmalloc.c:

```
void * vmalloc(unsigned long size)
```

The function returns a pointer to at least size bytes of virtually contiguous memory. On error, the function returns NULL.

To free an allocation obtained via vmalloc():

```
void vfree(const void *addr)
```

Slab Layer

To facilitate frequent allocations and deallocations of data, programmers often introduce free lists. A free list contains a block of available, already allocated, data structures:

- When code requires a new instance of a data structure, it can grab one of the structures off the free list rather than allocate the sufficient amount of memory and set it up for the data structure.
- When the data structure is no longer needed, it is returned to the free list instead of deallocated.

In this sense, the free list acts as an object cache, caching a frequently used type of object.

A main problem with free lists in the kernel is that there exists no global control. When available memory is low, there is no way for the kernel to communicate to every free list that it should shrink the sizes of its cache to free up memory. So, the Linux kernel provides the slab layer, which acts as generic data structure-caching layer.

The slab layer attempts to leverage several basic tenets:

- Frequently used data structures tend to be allocated and freed often, so cache them.
- Frequent allocation and deallocation can result in memory fragmentation. To prevent this, the cached free lists are arranged contiguously. Because freed data structures return to the free list, there is no resulting fragmentation.
- The free list provides improved performance during frequent allocation and deallocation because a freed object can be immediately returned to the next allocation.
- If the allocator is aware of concepts such as object size, page size, and total cache size, it can make more intelligent decisions.
- If part of the cache is made per-processor, allocations and frees can be performed without an SMP lock.
- If the allocator is NUMA-aware, it can fulfil allocations from the same memory node as the requestor.
- Stored objects can be coloured to prevent multiple objects from mapping to the same cache lines.

Design of the Slab Layer

- cache: The slab layer divides different objects into groups called caches, each of which stores a different type of object. There is one cache per object type.
- slab: The caches are then divided into slabs. The slabs are composed of one or more physical contiguous pages. Each cache may consist of multiple slabs.
- object: Each cache contains some number of objects, which are the data structures being called. Each slab is in one of three states: full, partial, or empty.

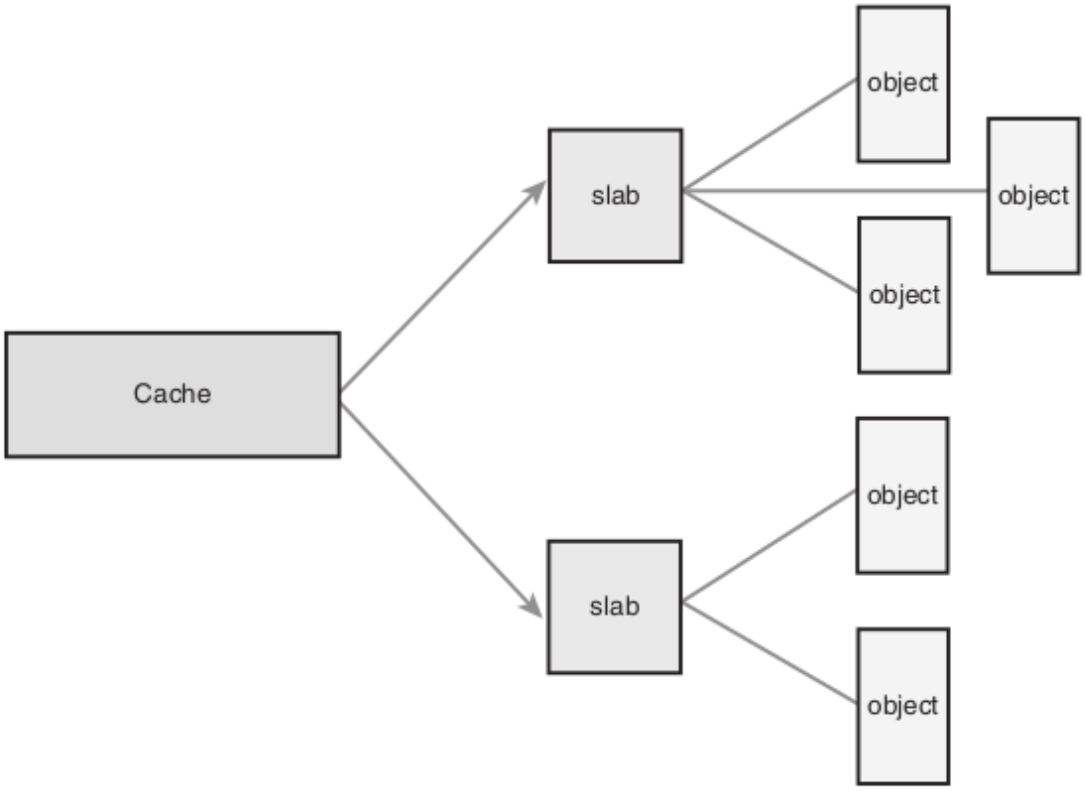

Figure 12.1 The relationship between caches, slabs, and objects.

Each cache is represented by a kmem_cache structure, which contains three lists, slabs_full, slabs_partial and slabs_empty (stored inside a kmem_list3 structure, which is defined in mm/slab.c). These lists contain all the slabs associated with the cache.

A slab descriptor, struct slab, represents each slab:

```
struct slab {
    struct list_head list;    /* full, partial, or empty list */
    unsigned long colouroff;  /* offset for the slab coloring */
    void *s_mem;              /* first object in the slab */
    unsigned int inuse;       /* allocated objects in the slab */
    kmem_bufctl_t free;       /* first free object, if any */
};
```

Slab descriptors are allocated either outside the slab in a general cache or inside the slab itself, at the beginning. The descriptor is stored inside the slab if the total size of the slab is sufficiently small, or if internal slack space is sufficient to hold the descriptor.

The slab allocator creates new slabs by interfacing with the low-level kernel page allocator via __get_free_pages():

```
static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
```

```
{
        struct page *page;
        void *addr;
        int i;

        flags |= cachep->gfpflags;
        if (likely(nodeid == -1)) {
                addr = (void*)__get_free_pages(flags, cachep->gfporder);
                if (!addr)
                   return NULL;
                page = virt_to_page(addr);
        } else {
                page = alloc_pages_node(nodeid, flags, cachep->gfporder);
                if (!page)
                   return NULL;
                addr = page_address(page);
        }

        i = (1 << cachep->gfporder);
        if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
                atomic_add(i, &slab_reclaim_pages);
        add_page_state(nr_slab, i);
        while (i--) {
                SetPageSlab(page);
                page++;
        }
        return addr;
}
```

This function uses __get_free_pages() to allocate memory sufficient to hold the cache. The first parameter points to the specific cache that needs more pages. The second parameter points to the flags given to __get_free_pages(). It adds default flags that the cache requires to the flags parameter. The third parameter node id makes the allocator NUMA-aware. When node id is not negative one,

the allocator attempts to fulfil the allocation from the same memory node that requested the allocation.

Memory is then freed by kmem_freepages(), which calls free_pages() on the given cache's pages. The freeing function is called only when available memory grows low and the system is attempting to free memory, or when a cache is explicitly destroyed.

Slab Allocator Interface

A new cache is created via kmem_cache_create():

struct kmem_cache * kmem_cache_create(const char *name, size_t size, size_t align, unsigned long flags, void (*ctor)(void *));

- The first parameter name is a string storing the name of the cache.
- The second parameter size is the size of each element in the cache.
- The third parameter align is the offset of the first object within a slab, which ensures a particular alignment within the page. A value of zero results in the standard alignment.
- The flags parameter specifies optional settings controlling the cache's behaviour. It can be zero, specifying no special behaviour, or one or more of the following flags OR'ed together:
    - SLAB_HWCACHE_ALIGN instructs the slab layer to align each object within a slab to a cache line, which prevents "false sharing" (two or more objects mapping to the same cache line despite existing at different addresses in memory). This improves performance but comes at a cost of increased memory footprint. For frequently used caches in performance-critical code, setting this option is a good idea.
    - SLAB_POISON causes the slab layer to fill the slab with a known value (a5a5a5a5). This is called poisoning and is useful for catching access to uninitialized memory.
    - SLAB_RED_ZONE causes the slab layer to insert "red zones" around the allocated memory to help detect buffer overruns.
    - SLAB_PANIC causes the slab layer to panic if the allocation fails. This flag is useful when the allocation must not fail.
    - SLAB_CACHE_DMA instructs the slab layer to allocate each slab in DMA-able memory. This is needed if the allocated object is used for DMA and must reside in ZONE_DMA.
- The final parameter ctor is a constructor for the cache. The constructor is called whenever new pages are added to the cache.

To destroy a cache:

int kmem_cache_destroy(struct kmem_cache *cachep)

The caller of this function must ensure two conditions before invoking this function:

- All slabs in the cache are empty.
- No one accesses the cache during and after a call to kmem_cache_destroy(). The caller must ensure this synchronization.

Allocating from the Cache

After a cache is created, an object is obtained from the cache via:

```
void * kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
```

This function returns a pointer to an object from the given cache cachep. If no free objects are in any slabs in the cache, and the slab layer must obtain new pages via kmem_getpages(), the value of flags is passed to __get_free_pages(). You probably want GFP_KERNEL or GFP_ATOMIC.

To later free an object and return it to its originating slab:

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp)
```

This marks the object objp in cachep as free.

Statically Allocating on the Stack

This is usually 8KB for 32-bit architectures and 16KB for 64-bit architectures because they usually have 4KB and 8KB pages.

Single-Page Kernel Stacks

When enabled the single-page kernel stacks option, each process is given only a single page. This was done for two reasons:

- It results in a page with less memory consumption per process.
- As uptime increases, it becomes increasingly hard to find two physically contiguous unallocated pages. Physical memory becomes fragmented, and the resulting VM pressure from allocating a single new process is expensive.

There is one more complication. Each process's entire call chain has to fit in its kernel stack. Historically, interrupt handlers also used the kernel stack of the process they interrupted, thus they too had to fit. This was efficient and simple, but it placed even tighter constraints on the already meagre kernel stack. When the stack moved to only a single page, interrupt handlers no longer fit. The kernel developers implemented interrupt stacks, which provide a single per-processor stack used for interrupt handlers.

Playing Fair on the Stack

Performing a large static allocation on the stack is dangerous. Therefore, it is wise to use a dynamic allocation scheme.

Hign Memory Mappings

By definition, pages in high memory might not be permanently mapped into the kernel's address space. Thus, pages obtained via alloc_pages() with the __GFP_HIGHMEM flag might not have a logical address.

Permanent Mappings

To map a given page structure into the kernel's address space, use this function, declared in <linux/highmem.h>:

```
void *kmap(struct page *page)
```

This function works on either high or low memory. If the page structure belongs to a page in low memory, the page's virtual address is simply returned. If the page resides in high memory, a permanent mapping is created and the address is returned. The function may sleep, so kmap() works only in process context.

Because the number of permanent mappings is limited, high memory should be unmapped when no longer needed. This is done via the following function, which unmaps the given page:

```
void kunmap(struct page *page)
```

Temporary Mappings

When a mapping must be created but the current context cannot sleep, the kernel provides temporary mappings. The kernel can atomically map a high memory page into one of the reserved mappings. Consequently, a temporary mapping can be used in places that cannot sleep, such as interrupt handlers, because obtaining the mapping never blocks.

Setting up a temporary mapping is done via:

```
void *kmap_atomic(struct page *page, enum km_type type)
```

The type parameter is one of the following enumerations defined in <asm-generic/kmap_types.h>, which describe the purpose of the temporary mapping.

```
enum km_type {
    KM_BOUNCE_READ,
    KM_SKB_SUNRPC_DATA,
    KM_SKB_DATA_SOFTIRQ,
    KM_USER0,
    KM_USER1,
    KM_BIO_SRC_IRQ,
    KM_BIO_DST_IRQ,
    KM_PTE0,
    KM_PTE1,
    KM_PTE2,
    KM_IRQ0,
    KM_IRQ1,
    KM_SOFTIRQ0,
    KM_SOFTIRQ1,
    KM_SYNC_ICACHE,
    KM_SYNC_DCACHE,
    KM_UML_USERCOPY,
```

```
        KM_IRQ_PTE,
        KM_NMI,
        KM_NMI_PTE,
        KM_TYPE_NR
    };
```

The mapping is undone via:

```
    void kunmap_atomic(void *kvaddr, enum km_type type)
```

This function also does not block. In many architectures it does not do anything at all except enable kernel pre-emption, because a temporary mapping is valid only until the next temporary mapping.

Per-CPU Allocations

Modern SMP-capable operating systems use per-CPU data (data that is unique to a given processor). Typically, per-CPU data is stored in an array. Each item in the array corresponds to a possible processor on the system. The current processor number indexes this array. You declare the data as:

```
    unsigned long my_percpu[NR_CPUS];
```

Then you can access it as:

```
    int cpu;
    cpu = get_cpu();  /* get current processor and disable kernel preemption */
    my_percpu[cpu]++; /* ... or whatever */
    printk("my_percpu on cpu=%d is %lu\n", cpu, my_percpu[cpu]);
    put_cpu();       /* enable kernel pre-emption */
```

There is no concurrency concerns exist and current processor can safely access the data without lock. And the call get_cpu() disables kernel pre-emption, where the corresponding call to put_cpu() enables kernel pre-emption.

The New percpu Interface

The header <linux/percpu.h> declares all the routines. You can find the actual definitions there, in mm/slab.c, and in <asm/percpu.h>.

Per-CPU Data at Compile-Time

Define a per-CPU variable at compile time:

```
    DEFINE_PER_CPU(type, name);
```

This creates an instance of a variable of type type, named name, for each processor on the system. If you need a declaration of the variable elsewhere, to avoid compile warnings, use the following macro:

```
    DECLARE_PER_CPU(type, name);
```

You can manipulate the variables with the get_cpu_var() and put_cpu_var() routines. A call to get_cpu_var() returns an lvalue for the given variable on the current processor. It also disables pre-emption, which put_cpu_var() correspondingly enables.

```
get_cpu_var(name)++; /* increment name on this processor */

put_cpu_var(name); /* done; enable kernel preemption */
```

You can obtain the value of another processor's per-CPU data, too:

```
per_cpu(name, cpu)++; /* increment name on the given processor */
```

You need to be careful with this approach because per_cpu() neither disables kernel pre-emption nor provides any sort of locking mechanism. The lockless nature of per-CPU data exists only if the current processor is the only manipulator of the data. If other processors touch other processors' data, you need locks.

Per-CPU Data at Runtime

The kernel implements a dynamic allocator, similar to kmalloc(), for creating per-CPU data. This routine creates an instance of the requested memory for each processor on the systems. The prototypes are in <linux/percpu.h>:

```
void *alloc_percpu(type); /* a macro */

void *__alloc_percpu(size_t size, size_t align);

void free_percpu(const void *);
```

The alloc_percpu() macro allocates one instance of an object of the given type for every processor on the system. It is a wrapper around __alloc_percpu(), which takes the actual number of bytes to allocate as a parameter and the number of bytes on which to align the allocation.
The alloc_percpu() macro aligns the allocation on a byte boundary that is the natural alignment of the given type. Such alignment is the usual behaviour.

A corresponding call to free_percpu() frees the given data on all processors.

A call to alloc_percpu() or __alloc_percpu() returns a pointer, which is used to indirectly reference the dynamically created per-CPU data. The kernel provides two macros to make this easy:

```
get_cpu_var(ptr);   /* return a void pointer to this processor's copy of ptr */

put_cpu_var(ptr);   /* done; enable kernel pre-emption */
```

The get_cpu_var() macro returns a pointer to the specific instance of the current processor's data. It also disables kernel pre-emption, which a call to put_cpu_var() then enables.

Reasons for Using Per-CPU Data

- The first is the reduction in locking requirements. Depending on the semantics by which processors access the per-CPU data, you might not need any locking at all.
- The second is per-CPU data greatly reduces cache invalidation. Because processors ideally access only their own data.

The only safety requirement for the use of per-CPU data is disabling kernel pre-emption.

Picking an Allocation Method

- kmalloc(): If you need contiguous physical pages.
- alloc_pages(): If you want to allocate from high memory.
- vmalloc(): If you do not need physically contiguous pages, only virtually contiguous.
- slab: If you are creating and destroying many large data structures.

13)

The Virtual Filesystem (VFS) is the subsystem of the kernel that implements the file and filesystem-related interfaces to user-space. All filesystems rely on the VFS not only to coexist but also to interoperate. This enables programs to use standard Unix system calls to read and write to different filesystems, even on different media.

Common Filesystem interfaces

The VFS enables system calls work between different filesystems and media. New filesystems and new Virtual of storage media can find their way into Linux, and programs need not be rewritten or even recompiled.

Filesystem Abstraction

Ther kernel implements an abstraction layer around its low-level filesystem interface. This abstraction layer works by defining the basic conceptual interfaces and data structures that all filesystems support.

Unix Filesystem

Unix has provided four basic filesystem-related abstractions:

- file: A file is an ordered string of bytes. The first byte marks the beginning of the file, and the last byte marks the end of the file. Each file is assigned a human-readable name for identification by both the system and the user.
- directory entries: Files are organized in directories. A directory is analogous to a folder and usually contains related files. Directories can also contain other directories, called subdirectories. In this fashion, directories may be nested to form paths. Each component of a path is called a directory entry, all of them called dentries. A directory is a file to the VFS.
- inodes: Information about a file called file metadata and is stored in a separate data structure form the file, called the inode.
- mount points: filesystems are mounted at a specific mount point in a global hierarchy known as a namespace. This enables all mounted filesystems to appear as entries in a single tree.

VFS Objects and Their Data Structures

The VFS is object-oriented. A family of data structures represents the common file model. These data structures are skin to objects.

The four primary object types of the VFS are:

- The superblock object, which represents a specific mounted filesystem.

- The inode object, which represents a specific file.
- The dentry object, which represents a directory entry, which is a single component of a path.
- The file object, which represents an open file are associated with a process.

An operations object is contained within each of these primary objects. These objects describe the methods that the kernel invokes against the primary objects:

- The super_operations object, which contains the methods that the kernel can invoke on a specific filesystem, such as write_inode() and sync_fs().
- The inode_operations object, which contains the methods that the kernel can invoke on a specific file, such as create() and link().
- The dentry_operations object, which contains the methods that the kernel can invoke on a specific directory entry, such as d_compare() and d_delete().
- The file_operations object, which contains the methods that a process can invoke on an open file, such as read() and write().

The operations objects are implemented as a structure of pointers to functions that operate on the parent object.

The Superblock Object

The superblock object is implemented by each filesystem and is used to store information describing that specific filesystem. This object usually corresponds to the filesystem superblock or the filesystem control block, which is stored in a special sector on disk. Filesystems that are not disk-based generate the superblock on-the-fly and store it in memory.

The superblock object is represented by struct super_block and defined in <linux/fs.h>:

```
struct super_block {
    struct list_head        s_list;         /* list of all superblocks */
    dev_t                   s_dev;          /* identifier */
    unsigned long           s_blocksize;    /* block size in bytes */
    unsigned char           s_blocksize_bits;   /* block size in bits */
    unsigned char           s_dirt;         /* dirty flag */
    unsigned long long      s_maxbytes;     /* max file size */
    struct file_system_type   s_type;       /* filesystem type */
    struct super_operations   s_op;         /* superblock methods */
    struct dquot_operations   *dq_op;       /* quota methods */
    struct quotactl_ops       *s_qcop;      /* quota control methods */
    struct export_operations  *s_export_op;   /* export methods */
    unsigned long           s_flags;        /* mount flags */
```

```
    unsigned long           s_magic;          /* filesystem's magic number */
    struct dentry           *s_root;          /* directory mount point */
    struct rw_semaphore     s_umount;         /* unmount semaphore */
    struct semaphore        s_lock;           /* superblock semaphore */
    int                     s_count;          /* superblock ref count */
    int                     s_need_sync;      /* not-yet-synced flag */
    atomic_t                s_active;         /* active reference count */
    void                    *s_security;      /* security module */
    struct xattr_handler    **s_xattr;        /* extended attribute handlers */
    struct list_head        s_inodes;         /* list of inodes */
    struct list_head        s_dirty;          /* list of dirty inodes */
    struct list_head        s_io;             /* list of writebacks */
    struct list_head        s_more_io;        /* list of more writeback */
    struct hlist_head       s_anon;           /* anonymous dentries */
    struct list_head        s_files;          /* list of assigned files */
    struct list_head        s_dentry_lru;     /* list of unused dentries */
    int                     s_nr_dentry_unused; /* number of dentries on list */
    struct block_device     *s_bdev;          /* associated block device */
    struct mtd_info         *s_mtd;           /* memory disk information */
    struct list_head        s_instances;      /* instances of this fs */
    struct quota_info       s_dquot;          /* quota-specific options */
    int                     s_frozen;         /* frozen status */
    wait_queue_head_t       s_wait_unfrozen;  /* wait queue on freeze */
    char                    s_id[32];         /* text name */
    void                    *s_fs_info;       /* filesystem-specific info */
    fmode_t                 s_mode;           /* mount permissions */
    struct semaphore        s_vfs_rename_sem;  /* rename semaphore */
    u32                     s_time_gran;      /* granularity of timestamps */
    char                    *s_subtype;       /* subtype name */
    char                    *s_options;       /* saved mount options */
};
```

The code for creating, managing, and destroying superblock objects lives in fs/super.c. A superblock object is created and initialized via the alloc_super() function. When mounted, a filesystem invokes this function, reads its superblock off of the disk, and fills in its superblock object.

Superblock Operations

The most important item in the superblock object is s_op, which is a pointer to the superblock operations table. The superblock operations table is represented by struct super_operations and is defined in <linux/fs.h>, which looks like this:

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, int);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    int (*show_options)(struct seq_file *, struct vfsmount *);
    int (*show_stats)(struct seq_file *, struct vfsmount *);
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
    int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

Each item in this structure is a pointer to a function that operates on a superblock object. The superblock operations perform low-level operations on the filesystem and its inodes.

When a filesystem needs to perform an operation on its superblock, it follows the pointers from its superblock object to the desired method. For example:

sb->s_op->write_super(sb);

The following are some of the superblock operations that are specified by super_operations:

- struct inode * alloc_inode(struct super_block *sb)
    - Creates and initializes a new inode object under the given superblock.
- void destroy_inode(struct inode *inode)
    - Deallocates the given inode.
- void dirty_inode(struct inode *inode)
    - Invoked by the VFS when an inode is dirtied (modified). Journaling filesystems such as ext3 and ext4 use this function to perform journal updates.
- void write_inode(struct inode *inode, int wait)
    - Writes the given inode to disk. The wait parameter specifies whether the operation should be synchronous.
- void drop_inode(struct inode *inode)
    - Called by the VFS when the last reference to an inode is dropped. Normal Unix filesystems do not define this function, in which case the VFS simply deletes the inode.
- void delete_inode(struct inode *inode)
    - Deletes the given inode from the disk.
- void put_super(struct super_block *sb)
    - Called by the VFS on unmount to release the given superblock object.The caller must hold the s_lock lock.
- void write_super(struct super_block *sb)
    - Updates the on-disk superblock with the specified superblock.The VFS uses this function to synchronize a modified in-memory superblock with the disk. The caller must hold the s_lock lock.
- int sync_fs(struct super_block *sb, int wait)
    - Synchronizes filesystem metadata with the on-disk filesystem. The wait parameter specifies whether the operation is synchronous.
- void write_super_lockfs(struct super_block *sb)
    - Prevents changes to the filesystem, and then updates the on-disk superblock with the specified superblock. It is currently used by LVM (the Logical Volume Manager).
- void unlockfs(struct super_block *sb)
    - Unlocks the filesystem against changes as done by write_super_lockfs().
- int statfs(struct super_block *sb, struct statfs *statfs)

- Called by the VFS to obtain filesystem statistics. The statistics related to the given filesystem are placed in statfs.

- int remount_fs(struct super_block *sb, int *flags, char *data)
  - Called by the VFS when the filesystem is remounted with new mount options.The caller must hold the s_lock lock.
- void clear_inode(struct inode *inode)
  - Called by the VFS to release the inode and clear any pages containing related data.
- void umount_begin(struct super_block *sb)
  - Called by the VFS to interrupt a mount operation. It is used by network filesystems, such as NFS.

The Inode Object

The inode Object represents all the information needed by the kernel to manipulate a file or directory. For Unix-style filesystems, this information is simply read from the on-disk inode.

The inode object is represented by struct inode and is defined in <linux/fs.h>.

```
struct inode {
    struct hlist_node      i_hash;          /* hash list */
    struct list_head       i_list;          /* list of inodes */
    struct list_head       i_sb_list;       /* list of superblocks */
    struct list_head       i_dentry;        /* list of dentries */
    unsigned long          i_ino;           /* inode number */
    atomic_t               i_count;         /* reference counter */
    unsigned int           i_nlink;         /* number of hard links */
    uid_t                  i_uid;           /* user id of owner */
    gid_t                  i_gid;           /* group id of owner */
    kdev_t                 i_rdev;          /* real device node */
    u64                    i_version        /* versioning number */
    loff_t                 i_size           /* file size in bytes */
    seqcount_t             i_size_seqcount;     /* serializer for i_size */
    struct timespec        i_atime;         /* last access time */
    struct timespec        i_mtime;         /* last modify time */
    struct timespec        i_ctime;         /* last change time */
    unsigned int           i_blkbits;       /* block size in bits */
```

```
    blkcnt_t            i_blocks;          /* block size in bits */
    unsigned short      i_bytes;           /* bytes consumed */
    umode_t             i_mode;            /* access permissions */
    spinlock_t          i_lock;            /* spinlock */
    struct rw_semaphore i_alloc_sem;       /* nests inside of i_sem */
    struct semaphore    i_sem;             /* inode semaphore */
    struct inode_operations *i_op;         /* inode ops table */
    struct file_operations  *i_fop;        /* default inode ops */
    struct super_block  *i_sb;             /* associated superblock */
    struct file_lock    *i_flock;          /* file lock list */
    struct address_space *i_mapping;       /* associated mapping */
    struct address_space i_data;           /* mapping for device */
    struct dquot        *i_dquot[MAXQUOTAS];  /* disk quotas for inode */
    struct list_head    i_devices;         /* list of block devices */
    union {
      struct pipe_inode_info  *i_pipe;     /* pipe information */
      struct block_device     *i_bdev;     /* block device driver */
      struct cdev             *i_cdev;     /* character device driver */
    };
    unsigned long       i_dnotify_mask;    /* directory notify mask */
    struct dnotify_struct   *i_dnotify;    /* dnotify */
    struct list_head    inotify_watches;   /* inotify watches */
    struct mutex        inotify_mutex;     /* protects inotify_watches */
    unsigned long       i_state;           /* state flags */
    unsigned long       dirtied_when;      /* first dirtying time */
    unsigned int        i_flags;           /* filesystem flags */
    atomic_t            i_writecount;      /* count of writers */
    void                *i_security;       /* security module */
    void                *i_private;        /* fs private pointer */
};
```

An inode represents each file on a filesystem, but the inode object is constructed in memory only as file are accessed.

Inode Operations

Inode Operations are invoked via:

```
i->i_op->truncate(i)
```

i is a reference to a particular inode, the truncate() operation defined by the filesystem on which i exists is called on the given inode.

The inode_operations structure is defined in <linux/fs.h>:

```
struct inode_operations {
   int (*create) (struct inode *, struct dentry *, int, struct nameidata *);
   struct dentry * (*lookup) (struct inode *, struct dentry *, struct nameidata *);
   int (*link) (struct dentry *, struct inode *, struct dentry *);
   int (*unlink) (struct inode *, struct dentry *);
   int (*symlink) (struct inode *, struct dentry *, const char *);
   int (*mkdir) (struct inode *, struct dentry *, int);
   int (*rmdir) (struct inode *, struct dentry *);
   int (*mknod) (struct inode *, struct dentry *, int, dev_t);
   int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
   int (*readlink) (struct dentry *, char __user *, int);
   void * (*follow_link) (struct dentry *, struct nameidata *);
   void (*put_link) (struct dentry *, struct nameidata *, void *);
   void (*truncate) (struct inode *);
   int (*permission) (struct inode *, int);
   int (*setattr) (struct dentry *, struct iattr *);
   int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
   int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
   ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
   ssize_t (*listxattr) (struct dentry *, const char *);
   int (*removexattr) (struct dentry *, const char *);
   void (*truncate_range) (struct inode *, loff_t, loff_t);
   long (*fallocate) (struct inode *inode, int mode, loff_t offset, loff_t len);
   int (*fiemap) (struct inode *, struct fiemap_extent_info *, u64 start, u64 len);
```

};

The following interfaces constitute the various functions that the VFS may perform, or ask a specific filesystem to perform, on a given inode:

- int create(struct inode *dir, struct dentry *dentry, int mode)
    - TheVFS calls this function from the creat() and open() system calls to create a new inode associated with the given dentry object with the specified initial access mode.
- struct dentry * lookup(struct inode *dir, struct dentry *dentry)
    - This function searches a directory for an inode corresponding to a filename specified in the given dentry.
- int link(struct dentry *old_dentry, struct inode *dir, struct dentry *dentry)
    - Invoked by the link() system call to create a hard link of the file old_dentry in the directory dir with the new filename dentry.
- int unlink(struct inode *dir, struct dentry *dentry)
    - Called from the unlink() system call to remove the inode specified by the directory entry dentry from the directory dir.
- int symlink(struct inode *dir, struct dentry *dentry, const char *symname)
    - Called from the symlink() system call to create a symbolic link named symname to the file represented by dentry in the directory dir.
- int mkdir(struct inode *dir, struct dentry *dentry, int mode)
    - Called from the mkdir() system call to create a new directory with the given initial mode.
- int rmdir(struct inode *dir, struct dentry *dentry)
    - Called by the rmdir() system call to remove the directory referenced by dentry from the directory dir.
- int mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t rdev)
    - Called by the mknod() system call to create a special file (device file, named pipe, or socket). The file is referenced by the device rdev and the directory entry dentry in the directory dir. The initial permissions are given via mode.
- int rename(struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry)
    - Called by the VFS to move the file specified by old_dentry from the old_dir directory to the directory new_dir, with the filename specified by new_dentry.
- int readlink(struct dentry *dentry, char *buffer, int buflen)
    - Called by the readlink() system call to copy at most buflen bytes of the full path associated with the symbolic link specified by dentry into the specified buffer.
- int follow_link(struct dentry *dentry, struct nameidata *nd)

    - Called by the VFS to translate a symbolic link to the inode to which it points. The link pointed at by dentry is translated, and the result is stored in the nameidata structure pointed at by nd.
- int put_link(struct dentry *dentry, struct nameidata *nd)
    - Called by the VFS to clean up after a call to follow_link().
- void truncate(struct inode *inode)
    - Called by the VFS to modify the size of the given file. Before invocation, the inode's i_size field must be set to the desired new size.
- int permission(struct inode *inode, int mask)
    - Checks whether the specified access mode is allowed for the file referenced by inode. This function returns zero if the access is allowed and a negative error code otherwise. Most filesystems set this field to NULL and use the generic VFS method, which simply compares the mode bits in the inode's objects to the given mask.
- int setattr(struct dentry *dentry, struct iattr *attr)
    - Called from notify_change() to notify a "change event" after an inode has been modified.
- int getattr(struct vfsmount *mnt, struct dentry *dentry, struct kstat *stat)
    - Invoked by the VFS upon noticing that an inode needs to be refreshed from disk. Extended attributes allow the association of key/values pairs with files.
- int setxattr(struct dentry *dentry, const char *name, const void *value, size_t size, int flags)
    - Used by the VFS to set the extended attribute name to the value value on the file referenced by dentry .
- ssize_t getxattr(struct dentry *dentry, const char *name, void *value, size_t size)
    - Used by the VFS to copy into value the value of the extended attribute name for the specified file.
- ssize_t listxattr(struct dentry *dentry, char *list, size_t size)
    - Copies the list of all attributes for the specified file into the buffer list.
- int removexattr(struct dentry *dentry, const char *name)
    - Removes the given attribute from the given file.

The Dentry Object

Dentry objects are all components in a path, including files.

Dentry objects are represented by struct dentry and defined in <linux/dcache.h>.

```
struct dentry {
  atomic_t          d_count;      /* usage count */
```

```
    unsigned int            d_flags;       /* dentry flags */
    spinlock_t              d_lock;        /* per-dentry lock */
    int                     d_mounted;     /* is this a mount point? */
    struct inode            *d_inode;      /* associated inode */
    struct hlist_node       d_hash;        /* list of hash table entries */
    struct dentry           *d_parent;     /* dentry object of parent */
    struct qstr             d_name;        /* dentry name */
    struct list_head        d_lru;         /* unused list */
    union {
      struct list_head      d_child;       /* list of dentries within */
      struct rcu_head       d_rcu;         /* RCU locking */
     } d_u;
    struct list_head        d_subdirs;     /* subdirectories */
    struct list_head        d_alias;       /* list of alias inodes */
    unsigned long           d_time;        /* revalidate time */
    struct dentry_operations  *d_op;       /* dentry operations table */
    struct super_block      *d_sb;         /* superblock of file */
    void                    *d_fsdata;     /* filesystem-specific data */
    unsigned char           d_iname[DNAME_INLINE_LEN_MIN];  /* short name */
};
```

Unlike the previous two objects, the dentry object does not correspond to any sort of on-disk data structure. The VFS creates it on-the-fly from a string representation of a path name. Because the dentry object is not physically stored on the disk, no flag in struct dentry specifies whether the object is modified.

Dentry State

A valid dentry object can be in one of three states:

- used: A used dentry corresponds to a valid inode and indicates that there are one or more users of the object.
- unused: An unused dentry corresponds to a valid inode, but the VFS is not currently using the dentry object.
- negative: A negative dentry is not associated with a valid inode because either the inode was deleted or the path name was resolved quickly.

The Dentry Cache

The kernel caches dentry objects in the dentry cache or, simply, the dcache. The dentry cache consists of three parts:

- List of "used" dentries linked off their associated inode via the i_dentry field of the inode object.
- A doubly linked "least recently used" list of unused and negative dentry objects.
- A hash table and hashing function used to quickly resolve a given path into the associated dentry object.

The hash table is represented by the dentry_hashtable array. Each element is a pointer to a list of dentries that hash to the same value. The size of this array depends on the amount of physical RAM in the system. The actual hash value is determined by d_hash)(). Hash table lookup is performed via d_lookup().

Dentry Operations

The dentry_operations structure specifies the methods that the VFS invokes on directory entries on a given filesystem. The dentry_operations structure is defined in <linux/dcache.h>:

```
struct dentry_operations {
    int (*d_revalidate) (struct dentry *, struct nameidata *);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete) (struct dentry *);
    void (*d_release) (struct dentry *);
    void (*d_iput) (struct dentry *, struct inode *);
    char *(*d_dname) (struct dentry *, char *, int);
};
```

The methods are as follows:

- int d_revalidate(struct dentry *dentry, struct nameidata *)
    - Determines whether the given dentry object is valid. The VFS calls this function whenever it is preparing to use a dentry from the dcache.
- int d_hash(struct dentry *dentry, struct qstr *name)
    - Creates a hash value from the given dentry. The VFS calls this function whenever it adds a dentry to the hash table.
- int d_compare(struct dentry *dentry, struct qstr *name1, struct qstr *name2)
    - Called by the VFS to compare two filenames, name1 and name2.
- int d_delete(struct dentry *dentry)

    - Called by the VFS when the specified dentry object's d_count reaches zero. This function requires the dcache_lock and the dentry's d_lock.
- void d_release(struct dentry *dentry)
    - Called by the VFS when the specified dentry is going to be freed.
- void d_iput(struct dentry *dentry, struct inode *inode)
    - Called by the VFS when a dentry object loses its associated inode.

The File Object

The file object is used to represent a file opened by a process. The file object is the in-memory representation of an open file. The object is created in response to the open() system call and destroyed in response to the close() system call. Because multiple processes can open and manipulate a file at the same time, there can be multiple file objects in existence for the same file. The file object merely represents a process's view of an open file. The object points back to the dentry that actually represents the open file. The inode and dentry objects are unique.

The file object is represented by struct file and is defined in <linux/sf.h>:

```
struct file {
    union {
      struct list_head       fu_list;          /* list of file objects */
      struct rcu_head        fu_rcuhead;       /* RCU list after freeing */
    } f_u;
    struct path              f_path;           /* contains the dentry */
    struct file_operations   *f_op;            /* file operations table */
    spinlock_t               f_lock;           /* per-file struct lock */
    atomic_t                 f_count;          /* file object's usage count */
    unsigned int             f_flags;          /* flags specified on open */
    mode_t                   f_mode;           /* file access mode */
    loff_t                   f_pos;            /* file offset (file pointer) */
    struct fown_struct       f_owner;          /* owner data for signals */
    const struct cred        *f_cred;          /* file credentials */
    struct file_ra_state     f_ra;             /* read-ahead state */
    u64                      f_version;        /* version number */
    void                     *f_security;      /* security module */
    void                     *private_data;    /* tty driver hook */
    struct list_head         f_ep_links;       /* list of epoll links */
```

```
    spinlock_t              f_ep_lock;      /* epoll lock */
    struct address_space    *f_mapping;     /* page cache mapping */
    unsigned long           f_mnt_write_state;  /* debugging state */
};
```

The file object does not actually correspond to any on-disk data. Therefore, no flag in the object represents whether the object is dirty and needs to be written back to disk. The file object does point to its associated dentry object via the f_dentry pointer. The dentry in turn points to the associated inode, which reflects whether the file itself is dirty.

File Operations

The file object methods are specified in file_operations and defined in <linux/fs.h>:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
            unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
```

```
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *,
                        int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area) (struct file *,
                                unsigned long,
                                unsigned long,
                                unsigned long,
                                unsigned long);
        int (*check_flags) (int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write) (struct pipe_inode_info *,
                                struct file *,
                                loff_t *,
                                size_t,
                                unsigned int);
        ssize_t (*splice_read) (struct file *,
                                loff_t *,
                                struct pipe_inode_info *,
                                size_t,
                                unsigned int);
        int (*setlease) (struct file *, long, struct file_lock **);
    };
```

- loff_t llseek(struct file *file, loff_t offset, int origin)
    - Updates the file pointer to the given offset. It is called via the llseek() system call.
- ssize_t read(struct file *file, char *buf, size_t count, loff_t *offset)
    - Reads count bytes from the given file at position offset into buf. The file pointer is then updated. This function is called by the read() system call.
- ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t offset)
    - Begins an asynchronous read of count bytes into buf of the file described in iocb. This function is called by the aio_read() system call.
- ssize_t write(struct file *file, const char *buf, size_t count, loff_t *offset)

    - Writes count bytes from buf into the given file at position offset. The file pointer is then updated. This function is called by the write() system call.
- ssize_t aio_write(struct kiocb *iocb, const char *buf, size_t count, loff_t offset)
    - Begins an asynchronous write of count bytes into buf of the file described in iocb. This function is called by the aio_write() system call.
- int readdir(struct file *file, void *dirent, filldir_t filldir)
    - Returns the next directory in a directory listing. This function is called by the readdir() system call.
- unsigned int poll(struct file *file, struct poll_table_struct *poll_table)
    - Sleeps, waiting for activity on the given file. It is called by the poll() system call.
- int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
    - Sends a command and argument pair to a device. It is used when the file is an open device node. This function is called from the ioctl() system call. Callers must hold the BKL.
- int unlocked_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
    - Implements the same functionality as ioctl() but without needing to hold the BKL. The VFS calls unlocked_ioctl() if it exists in lieu of ioctl() when user-space invokes the ioctl() system call. Thus filesystems need implement only one, preferably unlocked_ioctl().
- int compat_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
    - Implements a portable variant of ioctl() for use on 64-bit systems by 32-bit applications. This function is designed to be 32-bit safe even on 64-bit architectures, performing any necessary size conversions. New drivers should design their ioctl commands such that all are portable, and thus enable compat_ioctl() and unlocked_ioctl() to point to the same function. Like unlocked_ioctl(), compat_ioctl() does not hold the BKL.
- int mmap(struct file *file, struct vm_area_struct *vma)
    - Memory maps the given file onto the given address space and is called by the mmap() system call.
- int open(struct inode *inode, struct file *file)
    - Creates a new file object and links it to the corresponding inode object. It is called by the open() system call.
- int flush(struct file *file)
    - Called by the VFS whenever the reference count of an open file decreases. Its purpose is filesystem-dependent.
- int release(struct inode *inode, struct file *file)
    - Called by the VFS when the last remaining reference to the file is destroyed.

- int fsync(struct file *file, struct dentry *dentry, int datasync)
    - Called by the fsync() system call to write all cached data for the file to disk.
- int aio_fsync(struct kiocb *iocb, int datasync)
    - Called by the aio_fsync() system call to write all cached data for the file associated with iocb to disk.
- int fasync(int fd, struct file *file, int on)
    - Enables or disables signal notification of asynchronous I/O.
- int lock(struct file *file, int cmd, struct file_lock *lock)
    - Manipulates a file lock on the given file.
- ssize_t readv(struct file *file, const struct iovec *vector, unsigned long count, loff_t *offset)
    - Called by the readv() system call to read from the given file and put the results into the count buffers described by vector. The file offset is then incremented.
- ssize_t writev(struct file *file, const struct iovec *vector, unsigned long count, loff_t *offset)
    - Called by the writev() system call to write from the count buffers described by vector into the file specified by file. The file offset is then incremented.
- ssize_t sendfile(struct file *file, loff_t *offset, size_t size, read_actor_t actor, void *target)
    - Called by the sendfile() system call to copy data from one file to another. It performs the copy entirely in the kernel and avoids an extraneous copy to user-space.
- ssize_t sendpage(struct file *file, struct page *page, int offset, size_t size, loff_t *pos, int more)
    - Used to send data from one file to another.
- unsigned long get_unmapped_area(struct file *file, unsigned long addr, unsigned long len, unsigned long offset, unsigned long flags)
    - Gets unused address space to map the given file.
- int check_flags(int flags)
    - Used to check the validity of the flags passed to the fcntl() system call when the SETFL command is given.
- int flock(struct file *filp, int cmd, struct file_lock *fl)
    - Used to implement the flock() system call, which provides advisory locking.

Data Structures Associated with Filesystems

The kernel uses other standard data structures to manage data related to filesystems. The first object is used to describe a specific variant of a filesystem. The second data structure describes a mounted instance of a filesystem.

The file_system_type structure, defined in <linux/fs.h>:

```
struct file_system_type {
    const char          *name;      /* filesystem's name */
    int                 fs_flags;   /* filesystem type flags */

    /* the following is used to read the superblock off the disk */
    struct super_block  *(*get_sb) (struct file_system_type *, int,
                        char *, void *);

    /* the following is used to terminate access to the superblock */
    void                (*kill_sb) (struct super_block *);

    struct module       *owner;     /* module owning the filesystem */
    struct file_system_type *next;  /* next file_system_type in list */
    struct list_head    fs_supers;  /* list of superblock objects */

    /* the remaining fields are used for runtime lock validation */
    struct lock_class_key   s_lock_key;
    struct lock_class_key   s_umount_key;
    struct lock_class_key   i_lock_key;
    struct lock_class_key   i_mutex__key;
    struct lock_class_key   i_mutex__dir_key;
    struct lock_class_key   i_alloc_sem_key;
};
```

The get_sb() function reads the superblock from the disk and populates the superblock object when the filesystem is loaded. The remaining functions describe the filesystem's properties. There is only one file_system_type per filesystem, regardless of how many instances of the filesystem are mounted on the system, or whether the filesystem is even mounted at all.

When the filesystem is actually mounted, at which point the vfsmount structure is created. This structure represents a specific instance of a filesystem. The vfsmount structure is defined in <linux/mount.h>:

```
struct vfsmount {
    struct list_head    mnt_hash;       /* hash table list */
    struct vfsmount     *mnt_parent;    /* parent filesystem */
```

```
        struct dentry       *mnt_mountpoint;  /* dentry of this mount point */
        struct dentry       *mnt_root;       /* dentry of root of this fs */
        struct super_block   *mnt_sb;        /* superblock of this filesystem */
        struct list_head     mnt_mounts;     /* list of children */
        struct list_head     mnt_child;      /* list of children */
        int                  mnt_flags;      /* mount flags */
        char                 *mnt_devname;   /* device file name */
        struct list_head     mnt_list;       /* list of descriptors */
        struct list_head     mnt_expire;     /* entry in expiry list */
        struct list_head     mnt_share;      /* entry in shared mounts list */
        struct list_head     mnt_slave_list; /* list of slave mounts */
        struct list_head     mnt_slave;      /* entry in slave list */
        struct vfsmount      *mnt_master;    /* slave's master */
        struct mnt_namespace  *mnt_namespace;  /* associated namespace */
        int                  mnt_id;         /* mount identifier */
        int                  mnt_group_id;   /* peer group identifier */
        atomic_t             mnt_count;      /* usage count */
        int                  mnt_expiry_mark;  /* is marked for expiration */
        int                  mnt_pinned;     /* pinned count */
        int                  mnt_ghosts;     /* ghosts count */
        atomic_t             __mnt_writers;  /* writers count */
    };
```

The complicated part of maintaining the list of all mount points is the relation between the filesystem and all the other mount points. The various linked lists in vfsmount keep track of this information. The vfsmount structure also stores the flags, if any, specified on mount in the mnt_flags field.

Flag	**Description**
MNT_NOSUID	Forbids setuid and setgid flags on binaries on this filesystem
MNT_NODEV	Forbids access to device files on this filesystem
MNT_NOEXEC	Forbids execution of binaries on this filesystem

Data Structures Associated with a Process

Each process on the system has its own list of open files, root filesystem, current working directory, mount points, and so on. Three data structures tie together the VFS layer and the processes on the system: files_struct, fs_struct, and namespace.

The files_struct is defined in <linux/fdtable.h>. This table's address is pointed to by the files entry in the processor descriptor. All per-process information about open files and file descriptors is contained therein:

```
struct files_struct {
    atomic_t             count;          /* usage count */
    struct fdtable       *fdt;           /* pointer to other fd table */
    struct fdtable       fdtab;          /* base fd table */
    spinlock_t           file_lock;      /* per-file lock */
    int                  next_fd;        /* cache of next available fd */
    struct embedded_fd_set  close_on_exec_init; /* list of close-on-exec fds */
    struct embedded_fd_set  open_fds_init      /* list of open fds */
    struct file          *fd_array[NR_OPEN_DEFAULT]; /* base files array */
};
```

The array fd_array points to the list of open file objects. If a process opens more than 64 file objects, the kernel allocates a new array and points the fdt pointer at it.

The second process-ralated structure is fs_struct, which contains filesystem information related to a process and is pointed at by the fs field in the process descriptor. The structure is defined in <linux/fs_struct.h>.

```
struct fs_struct {
    int         users;    /* user count */
    rwlock_t    lock;     /* per-structure lock */
    int         umask;    /* umask */
    int         in_exec;  /* currently executing a file */
    struct path   root;     /* root directory */
    struct path   pwd;      /* current working directory */
};
```

This structure holds the current working directory and root directory of the current process.

The third and final structure is the namespace structure, which is defined in <linux/mnt_namespace.h> and pointed at by the mnt_namespace field in the process descriptor.

```
struct mnt_namespace {
```

```
        atomic_t          list;   /* usage count */
        struct vfsmount     *root;  /* root directory */
        struct list_head    list;   /* list of mount points */
        wait_queue_head_t   poll;   /* polling waitqueue */
        int               event;  /* event count */
};
```

The list member specifies a doubly linked list of the mounted filesystems that make up the namespace.

14)

Block devices are hardware devices distinguished by the random access of fixed-size chunks of data. The fixed-size chunks of data are called blocks.

The other basic type of device is a character device. Character devices, or char devices, are accessed as a stream of sequential data, one byte after another.

Anatomy of a Block Device

The smallest addressable unit on a block device is a sector. Sectors come in various powers of two, but 512 bytes is the most common size. The sector size is a physical property of the device, and the sector is the fundamental unit of all block devices.

block is the smallest logically addressable unit. The block is an abstraction of the filesystem, filesystems can be accessed only in multiples of a block.

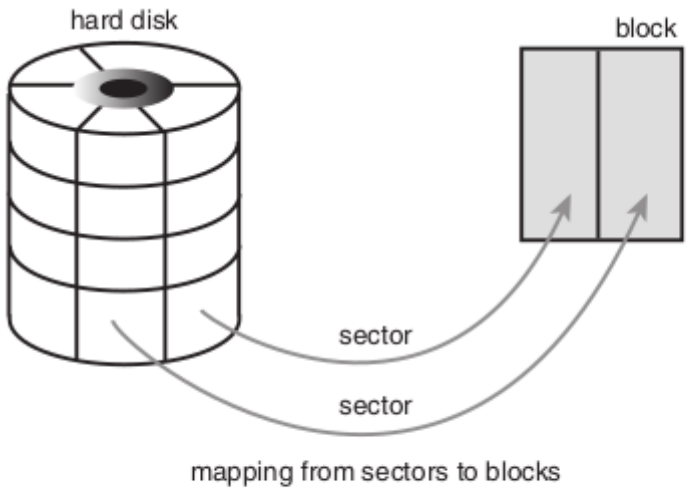

Figure 14.1 Relationship between sectors and blocks.

Buffers and Buffer Heads

When a block is stored in memory, it is stored in a buffer. Each buffer is associated with exactly one block. The buffer serves as the object that represents a disk block in memory. Because the kernel requires some associated control information to accompany the data, each buffer is associated with

a descriptor. This descriptor is called a buffer head and is of type struct buffer_head.
The buffer_head structure holds all the information that the kernel needs to manipulate buffers and is defined in <linux/buffer_head.h>.

```
struct buffer_head {
   unsigned long b_state;           /* buffer state flags */
   struct buffer_head *b_this_page;   /* list of page's buffers */
   struct page *b_page;             /* associated page */
   sector_t b_blocknr;              /* starting block number */
   size_t b_size;                   /* size of mapping */
   char *b_data;                    /* pointer to data within the page */
   struct block_device *b_bdev;       /* associated block device */
   bh_end_io_t *b_end_io;           /* I/O completion */
   void *b_private;                 /* reserved for b_end_io */
   struct list_head b_assoc_buffers;  /* associated mappings */
   struct address_space *b_assoc_map; /* associated address space */
   atomic_t b_count;                /* use count */
};
```

The b_state field specifies the state of this particular buffer. It can be one or more of the flags in the following table. The legal flags are stored in the bh_state_bits enumeration, which is defined in <linux/buffer_head.h>.

Status Flag	**Meaning**
BH_Uptodate	Buffer contains valid data.
BH_Dirty	Buffer is dirty. (The contents of the buffer are newer than the contents of the block on disk and therefore the buffer must eventually be written back to disk.)
BH_Lock	Buffer is undergoing disk I/O and is locked to prevent concurrent access.
BH_Req	Buffer is involved in an I/O request.
BH_Mapped	Buffer is a valid buffer mapped to an on-disk block.
BH_New	Buffer is newly mapped via get_block() and not yet accessed.

Status Flag	Meaning
BH_Async_Read	Buffer is undergoing asynchronous read I/O via end_buffer_async_read().
BH_Async_Write	Buffer is undergoing asynchronous write I/O via end_buffer_async_write().
BH_Delay	Buffer does not yet have an associated on-disk block (delayed allocation).
BH_Boundary	Buffer forms the boundary of contiguous blocks; the next block is discontinuous.
BH_Write_EIO	Buffer incurred an I/O error on write.
BH_Ordered	Ordered write.
BH_Eopnotsupp	Buffer incurred a "not supported" error.
BH_Unwritten	Space for the buffer has been allocated on disk but the actual data has not yet been written out.
BH_Quiet	Suppress errors for this buffer.

The bh_state_bits enumeration also contains a BH_PrivateStart flag. This is not a valid state flag but instead corresponds to the first usable bit of which other code can make use. All bit values equal to and greater than BH_PrivateStart are not used by the block I/O layer proper, so these bits are safe to use by individual drivers who want to store information in the b_state field.

The b_count field is the buffer's usage count. The value is incremented and decremented by two inline functions defined in <linux/buffer_head.h>:

```
static inline void get_bh(struct buffer_head *bh)
{
    atomic_inc(&bh->b_count);
}

static inline void put_bh(struct buffer_head *bh)
{
    smp_mb__before_atomic_dec();
    atomic_dec(&bh->b_count);
}
```

Before manipulating a buffer head, you must increment its reference count via get_bh() to ensure that the buffer head is not deallocated out from under you. When finished with the buffer head, decrement the reference count via put_bh().

The physical block on disk to which a given buffer corresponds is the b_blocknr-th logical block on the block device described by b_bdev. The physical page in memory to which a given buffer corresponds is the page pointed to by b_page. More specifically, b_data is a pointer directly to the block (that exists somewhere in b_page), which is b_size bytes in length. Therefore, the block is located in memory starting at address b_data and ending at address (b_data + b_size).

The purpose of a buffer head is to describe this mapping between the on-disk block and the physical in-memory buffer. Acting as a descriptor of this buffer-to-block mapping is the data structure's only role in the kernel.

The bio Structure

The bio structure is the basic container for block I/O within the kernel is the bio structure. Defined in <linux/bio.h>, this structure represents block I/O operations that are in flight as a list of segments. A segment is a chunk of a buffer that is contiguous in memory. Thus, individual buffers need not be contiguous in memory. Vector I/O such as this is called scatter-gather I/O.

```
struct bio {
    sector_t bi_sector;             /* associated sector on disk */
    struct bio *bi_next;            /* list of requests */
    struct block_device *bi_bdev;   /* associated block device */
    unsigned long bi_flags;         /* status and command flags */
    unsigned long bi_rw;            /* read or write? */
    unsigned short bi_vcnt;         /* number of bio_vecs off */
    unsigned short bi_idx;          /* current index in bi_io_vec */
    unsigned short bi_phys_segments;  /* number of segments */
    unsigned int bi_size;           /* I/O count */
    unsigned int bi_seg_front_size;   /* size of first segment */
    unsigned int bi_seg_back_size;    /* size of last segment */
    unsigned int bi_max_vecs;       /* maximum bio_vecs possible */
    unsigned int bi_comp_cpu;       /* completion CPU */
    atomic_t bi_cnt;                /* usage counter */
    struct bio_vec *bi_io_vec;      /* bio_vec list */
    bio_end_io_t *bi_end_io;        /* I/O completion method */
    void *bi_private;               /* owner-private method */
    bio_destructor_t *bi_destructor;  /* destructor method */
```

```
    struct bio_vec bi_inline_vecs[0]; /* inline bio vectors */
};
```

The primary purpose of a bio structure is to represent an in-flight block I/O operation. To this end, the majority of the fields in the structure are housekeeping related. The most important fields are bi_io_vec, bi_vcnt, and bi_idx.

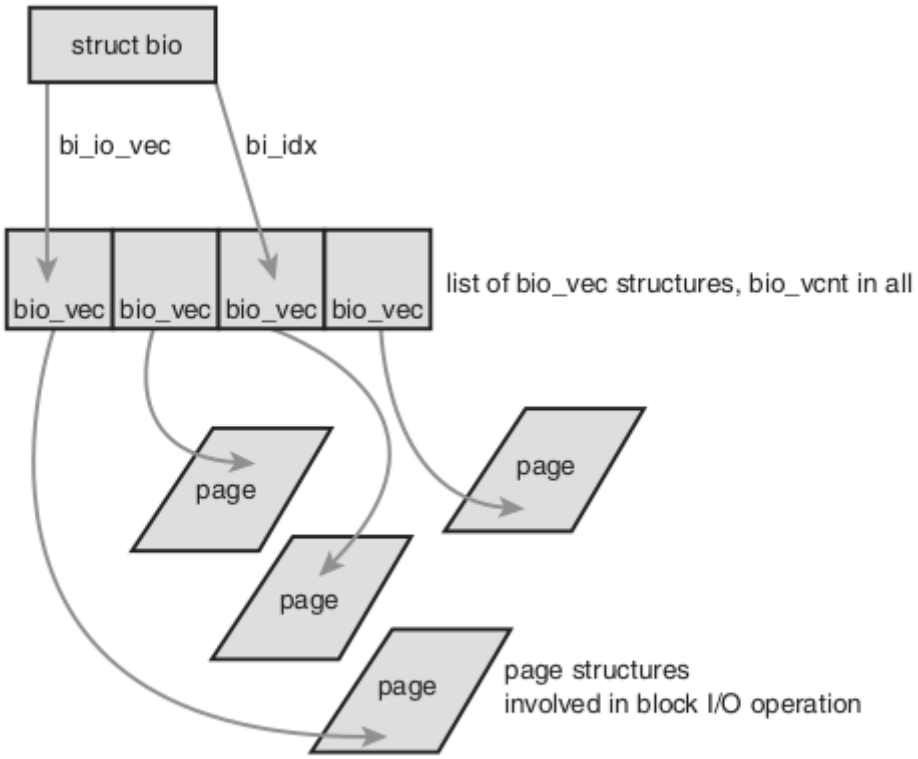

Figure 14.2 Relationship between **struct bio**, **struct bio_vec**, and **struct page**.

I/O vectors

The bi_io_vec field points to an array of bio_vec structures, each of which is used as a list of individual segments in this specific block I/O operation. The entire array of these vectors describes the entire buffer.

Each bio_vec is treated as a vector of the form <page, offset, len>, which describes a specific segment: the physical page on which it lies, the location of the block as an offset into the page, the length of the block starting from the given offset.

The bio_vec structure is defined in <linux/bio.h>:

```
struct bio_vec {
    /* pointer to the physical page on which this buffer resides */
    struct page *bv_page;
```

```
    /* the length in bytes of this buffer */
    unsigned int bv_len;

    /* the byte offset within the page where the buffer resides */
    unsigned int bv_offset;
};
```

In each given block I/O operation, there are bi_vcnt vectors in the bio_vec array starting with bi_io_vec. As the block I/O operation is carried out, the bi_idx field is used to point to the current index into the array.

Each block I/O request is represented by a bio structure. Each request is composed of one or more blocks, which are stored in an array of bio_vec structures. These structures act as vectors and describe each segment's location in a physical page in memory. The first segment in the I/O operation is pointed to by b_io_vec. Each additional segment follows after the first, for a total of bi_vcnt segments in the list. As the block I/O layer submits segments in the request, the bi_idx field is updated to point to the current segment.

The bi_idx field is used to point to the current bio_vec in the list, which helps the block I/O layer keep track of partially completed block I/O operations. More importantly, it allows the splitting of bio structures.

The bio structure maintains a usage count in the bi_cnt field. When this field reaches zero, the structure is destroyed and the backing memory is freed.

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

The bi_private field is a private field for the owner of the structure. As a rule, you can read or write this field only if you allocated the bio structure.

Request Queues

Block devices maintain request queues to store their pending block I/O requests. The request queue is represented by the request_queue structure and is defined in <linux/blkdev.h>. The request queue contains a doubly linked list of requests and associated control information. Requests are added to the queue by higher-level code in the kernel, such as filesystems. As long as the request queue is nonempty, the block device driver associated with the queue grabs the request from the head of the queue and submits it to its associated block device. Each item in the queue's request list is a single request, of type struct request (also defined in <linux/blkdev.h>).

Each request can be composed of more than one bio structure because individual requests can operate on multiple consecutive disk blocks. Note that although the blocks on the disk must be adjacent, the blocks in memory need not be; each bio structure can describe multiple segments and the request can be composed of multiple bio structures.

I/O Schedulers

The subsystem of the kernel that performs these operations is called the I/O scheduler. The I/O scheduler divides the resource of disk I/O among the pending block I/O requests in the system.

The Job of an I/O Scheduler

An I/O scheduler works by managing a block device's request queue. It decides the order of requests in the queue and at what time each request is dispatched to the block device. It manages the request queue with the goal of reducing seeks, which results in greater global throughput.

I/O scheduler perform two primary actions to minimize seeks:

- merging: the coalescing of two or more requests into one.
- sorting: the entire request queue is kept sorted, sectorwise.

I/O Scheduler Selection

There are four different I/O schedulers in the 2.6 kernel. Each of these I/O schedulers can be enabled and built into the kernel. By default, block devices use the Complete Fair Queuing I/O scheduler. This can be overridden via the boot-time option elevator=foo on the kernel command line, where foo is a valid and enabled I/O Scheduler.

Parameter	I/O Scheduler
as	Anticipatory
cfq	Complete Fair Queuing
deadline	Deadline
noop	Noop

15)

The kernel also has to manage the memory of user-space process. This memory is called the process address space, which is the representation of memory given to each user-space process on the system. Linux is a virtual memory operating system, and thus the resource of memory is virtualized among the processes on the system.

Address Space

The process address space consists of the virtual memory addressable by a process and the addresses within the virtual memory that the process is allowed to use. Each process is given a flat address space that is unique to each process. A memory address in another process's address space. Both processes can have different data at the same address space in their respective address space. Alternatively, processes can elect to share their address space with other processes. We know these processes as threads.

A memory address is a given value within the address space. Intervals of legal address are called memory areas. The process, through the kernel, can dynamically add and remove memory areas to its address space.

The process can access a memory address only in a valid memory area. Memory areas have associated permissions, such as readable, writable, and executable, that the associated process must respect.

Memory areas can contain all sorts of goodies:

- A memory map of the executable file's code, called the text section.
- A memory map of the executable file's initialized global variables, called the data section.
- A memory map of the zero page containing uninitialized global variables, called the bss section.
- A memory map of the zero page used for the process's user-space stack.
- An additional text, data, and bss section for each shared library, such as the C library and dynamic linker, loaded into the process's address space.
- Any memory mapped files.
- Any shared memory segments.
- Any anonymous memory mappings, such as those associated with malloc().

All valid addresses in the process address space exist in exactly one area; memory areas do not overlap.

The Memory Descriptor

The kernel represents a process's address space with a data structure called the memory descriptor. This structure contains all the information related to the process address space.

The memory descriptor is represented by struct mm_struct and defined in <linux/mm_types.h>:

```
struct mm_struct {
   struct vm_area_struct    *mmap;          /* list of memory areas */
   struct rb_root           mm_rb;          /* red-black tree of VMAs */
   struct vm_area_struct    *mmap_cache;    /* last used memory area */
   unsigned long            free_area_cache;  /* 1st address space hole */
   pgd_t                    *pgd;           /* page global directory */
   atomic_t                 mm_users;       /* address space users */
   atomic_t                 mm_count;       /* primary usage counter */
   int                      map_count;      /* number of memory areas */
   struct rw_semaphore      mmap_sem;       /* memory area semaphore */
   spinlock_t               page_table_lock;  /* page table lock */
   struct list_head         mmlist;         /* list of all mm_structs */
   unsigned long            start_code;     /* start address of code */
```

```
    unsigned long           end_code;       /* final address of code */
    unsigned long           start_data;     /* start address of data */
    unsigned long           end_data;       /* final address of data */
    unsigned long           start_brk;      /* start address of heap */
    unsigned long           brk;            /* final address of heap */
    unsigned long           start_stack;    /* start address of stack */
    unsigned long           arg_start;      /* start of arguments */
    unsigned long           arg_end;        /* end of arguments */
    unsigned long           env_start;      /* start of environment */
    unsigned long           env_end;        /* end of environment */
    unsigned long           rss;            /* pages allocated */
    unsigned long           total_vm;       /* total number of pages */
    unsigned long           locked_vm;      /* number of locked pages */
    unsigned long           saved_auxv[AT_VECTOR_SIZE]; /* saved auxv */
    cpumask_t               cpu_vm_mask;    /* lazy TLB switch mask */
    mm_context_t            context;        /* arch-specific data */
    unsigned long           flags;          /* status flags */
    int                     core_waiters;   /* thread core dump waiters */
    struct core_state       *core_state;    /* core dump support */
    spinlock_t              ioctx_lock;     /* AIO I/O list lock */
    struct hlist_head       ioctx_list;     /* AIO I/O list */
};
```

Allocating a Memory Descriptor

The memory descriptor associated with a given task is stored in the mm field of the task's process descriptor. Thus, current->mm is the current process's memory descriptor. The copy_mm() function copies a parent's memory descriptor to its child during fork(). The mm_struct structure is allocated from the mm_cachep slab cache via the allocate_mm() macro in kernel/fork.c. Normally, each process receives a unique mm_struct and thus a unique process address space.

Processes may share their address spaces with their children via the CLONE_VM flag to clone(). The process is then called a thread. This is essentially the only difference between normal processes and so-called threads in Linux; the Linux kernel does not otherwise differentiate between them. Threads are regular processes to the kernel that merely share certain resources. When CLONE_VM is specified, allocate_mm() is not called, and the process's mm field is set to point to the memory descriptor of its parent.

Destroying a Memory Descriptor

When the process associated with a specific address space exits, exit_mm(), defined in kernel/exit.c, is invoked. This function performs some housekeeping and updates some statistics. It then calls mmput(), which decrements the memory descriptor's mm_users user counter. If the user count reaches zero, mmdrop() is called to decrement the mm_count usage counter. If that counter is finally zero, the free_mm() macro is invoked to return the mm_struct to the mm_cachep slab cache via kmem_cache_free(), because the memory descriptor does not have any users.

The mm_struct and Kernel Threads

Kernel threads do not have a process address space and therefore do not have an associated memory descriptor. Thus, the mm field of a kernel thread's process descriptor is NULL. This is the definition of a kernel thread: processes that have no user context.

Kernel threads do not ever access any user-space memory and do not have any pages in user-space, they do not deserve their own memory descriptor and page tables. Despite this, kernel threads need some of the data, such as the page tables, even to access kernel memory. To provide kernel threads the needed data, without wasting memory on a memory descriptor and page tables, or wasting processor cycles to switch to a new address space whenever a kernel thread begins running, kernel threads use the memory descriptor of whatever task ran previously.

Whenever a process is scheduled, the process address space referenced by the process's mm field is loaded. The active_mm field in the process descriptor is then updated to refer to the new address space. Kernel threads do not have an address space and mm is NULL. Therefore, when a kernel thread is scheduled, the kernel notices that mm is NULL and keeps the previous process's address space loaded. The kernel then updates the active_mm field of the kernel thread's process descriptor to refer to the previous process's memory descriptor. The kernel thread can then use the previous process's page tables as needed. Because kernel threads do not access user-space memory, they make use of only the information in the address space pertaining to kernel memory, which is the same for all processes.

Virtual Memory Areas

The memory area structure, vm_area_struct, represents memory areas. It is defined in <linux/mm_types.h>. In the Linux kernel, memory areas are often called virtual memory areas (VMAs).

The vm_area_struct structure describes a single memory area over a contiguous interval in a given address space. The kernel treats each memory area as a unique memory object. Each memory area prossesses certain properties. In this manner, each VMA structure can represent different types of memory areas.

```
struct vm_area_struct {
    struct mm_struct        *vm_mm;         /* associated mm_struct */
    unsigned                long vm_start;  /* VMA start, inclusive */
    unsigned long           vm_end;         /* VMA end , exclusive */
    struct vm_area_struct   *vm_next;       /* list of VMA's */
    pgprot_t                vm_page_prot;   /* access permissions */
```

```
    unsigned long          vm_flags;      /* flags */
    struct rb_node         vm_rb;         /* VMA's node in the tree */
    union { /* links to address_space->i_mmap or i_mmap_nonlinear */
    struct {
         struct list_head       list;
         void                   *parent;
         struct vm_area_struct   *head;
       } vm_set;
       struct prio_tree_node prio_tree_node;
    } shared;
    struct list_head            anon_vma_node;    /* anon_vma entry */
    struct anon_vma             *anon_vma;       /* anonymous VMA object */
    struct vm_operations_struct   *vm_ops;         /* associated ops */
    unsigned long               vm_pgoff;        /* offset within file */
    struct file                 *vm_file;        /* mapped file, if any */
    void                        *vm_private_data; /* private data */
};
```

Each memory descriptor is associated with a unique interval in the process's address space. Intervals in different memory areas in the same address space cannot overlap. The vm_mm field points to this VMA's associated mm_struct.

VMA Flags

The vm_flags field contains bit flags, defined in <linux/mm.h>, that specify the behavior of and provide information about the pages contained in the memory area. vm_flags contains information that relates to the memory area as a whole (each page), not specific individual pages.

Flag	Effect on the VMA and Its Pages
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.

Flag	Effect on the VMA and Its Pages
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.
VM_GROWSDOWN	The area can grow downward.
VM_GROWSUP	The area can grow upward.
VM_SHM	The area is used for shared memory.
VM_DENYWRITE	The area maps an unwritable file.
VM_EXECUTABLE	The area maps an executable file.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages seem to be accessed sequentially.
VM_RAND_READ	The pages seem to be accessed randomly.
VM_DONTCOPY	This area must not be copied on fork().
VM_DONTEXPAND	This area cannot grow via mremap().
VM_RESERVED	This area must not be swapped out.
VM_ACCOUNT	This area is an accounted VM object.
VM_HUGETLB	This area uses hugetlb pages.
VM_NONLINEAR	This area is a nonlinear mapping.

VMA Operations

The vm_ops field in the vm_area_struct structure points to the table of operations associated with a given memory area, which the kernel can invoke to manipulate the VMA. The vm_area_struct acts as a generic object for representing any type of memory area, and the operations table describes the specific methods that can operate on this particular instance of the object.

The operations table is represented by struct vm_operations_struct and is defined in <linux/mm.h>:

```
struct vm_operations_struct {
  void (*open) (struct vm_area_struct *);
  void (*close) (struct vm_area_struct *);
  int (*fault) (struct vm_area_struct *, struct vm_fault *);
  int (*page_mkwrite) (struct vm_area_struct *vma, struct vm_fault *vmf);
  int (*access) (struct vm_area_struct *, unsigned long ,
  void *, int, int);
};
```

- void open(struct vm_area_struct *area)
    - This function is invoked when the given memory area is added to an address space.
- void close(struct vm_area_struct *area)
    - This function is invoked when the given memory area is removed from an address space.
- int fault(struct vm_area_sruct *area, struct vm_fault *vmf)
    - This function is invoked by the page fault handler when a page that is not present in physical memory is accessed.
- int page_mkwrite(struct vm_area_sruct *area, struct vm_fault *vmf)
    - This function is invoked by the page fault handler when a page that was read-only is being made writable.
- int access(struct vm_area_struct *vma, unsigned long address, void *buf, int len, int write)
    - This function is invoked by access_process_vm() when get_user_pages() fails.

Lists and Trees of Memory Areas

Memory areas are accessed via both the mmap and the mm_rb field of the memory descriptor. These two data structures independently point to all the memory area objects associated with the memory descriptor. In fact, they both contain pointers to the same vm_area_struct structures, merely represented in different ways.

- mmap: links together all the memory area objects in a single linked list.
  Each vm_area_struct structure is linked into the list via its vm_next field. The areas are sorted by ascending address. The first memory area is the vm_area_struct structure to which mmap points.The last structure points to NULL.
- mm_rb: links together all the memory area objects in a red-black tree. The root of the red-black tree is mm_rb, and each vm_area_struct structure in this address space is linked to the tree via its vm_rb field.

The linked list is used when every node needs to be traversed. The red-black tree is used when locating a specific memory area in the address space.

Manipulating Memory Areas

The kernel often has to perform operations on a memory area, these operations are frequent and form the basis of the mmap() routine. Helper function are defined to assist these jobs. These functions are all declared in <linux/mm.h>.

find_vma()

The kernel provides a function find_vma() for searching for the VMA in which a given memory address resides. It is defined in mm/mmap.c:

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr)
```

This function searches the given address space for the first memory area whose vm_end field is greater than addr. The result of the find_vma function is cached in the mmap_cache field of the memory descriptor.

If the given address is not in the cache, you must search the memory areas associated with this memory descriptor for a match. This is done via the red-black tree:

```
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
        struct vm_area_struct *vma = NULL;

        if (mm) {
                /* Check the cache first. */
                /* (Cache hit rate is typically around 35%.) */
                vma = mm->mmap_cache;
                if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
                        struct rb_node * rb_node;

                        rb_node = mm->mm_rb.rb_node;
                        vma = NULL;

                        while (rb_node) {
                                struct vm_area_struct * vma_tmp;

                                vma_tmp = rb_entry(rb_node,
```

```
                                        struct vm_area_struct, vm_rb);

                        if (vma_tmp->vm_end > addr) {
                                vma = vma_tmp;
                                if (vma_tmp->vm_start <= addr)
                                        break;
                                rb_node = rb_node->rb_left;
                        } else
                                rb_node = rb_node->rb_right;
                }
                if (vma)
                        mm->mmap_cache = vma;
        }
}
return vma;
```

find_vma_prev()

The find_vma_prev() function works the same as find_vma(), but it also returns the last VMA before addr. The function is also defined in mm/mmap.c and declared in <linux/mm.h>:

```
struct vm_area_struct * find_vma_prev(struct mm_struct *mm, unsigned long addr, struct vm_area_struct **pprev)
```

The pprev argument stores a pointer to the VMA preceding addr.

find_vma_intersection()

The find_vma_intersection() function returns the first VMA that overlaps a given address interval. The function is defined in <linux/mm.h> because it is inline:

```
static inline struct vm_area_struct *
find_vma_intersection(struct mm_struct *mm, unsigned long start_addr, unsigned long end_addr)
{
   struct vm_area_struct *vma;


   vma = find_vma(mm, start_addr);
   if (vma && end_addr <= vma->vm_start)
```

```
    vma = NULL;
    return vma;
}
```

The first parameter is the address space to search, start_addr is the start of the interval, and end_addr is the end of the interval.

mmap() and do_mmap(): Creating an Address Interval

do_mmap() is the function used to add an address interval to a process's address space, whether that means expanding an existing memory area or creating a new one.

The do_mmap() function is defined in <linux/mm.h>:

```
unsigned long do_mmap(struct file *file, unsigned long addr, unsigned long len, unsigned long prot, unsigned long flag, unsigned long offset)
```

This function maps the file specified by file at offset offset for length len. The file parameter can be NULL and offset can be zero, in which case the mapping will not be backed by a file. In that case, this is called an anonymous mapping. If a file and offset are provided, the mapping is called a file-backed mapping.

The addr function optionally specifies the initial address from which to start the search for a free interval.

The prot parameter specifies the access permissions for pages in the memory area. The possible permission flags are defined in <asm/mman.h> and are unique to each supported architecture, although in practice each architecture defines the flags listed in the following table:

Flag	Effect on the Pages in the New Interval
PROT_READ	Corresponds to VM_READ
PROT_WRITE	Corresponds to VM_WRITE
PROT_EXEC	Corresponds to VM_EXEC
PROT_NONE	Cannot access page

The flags parameter specifies flags that correspond to the remaining VMA flags. These flags specify the type and change the behavior of the mapping. They are also defined in <asm/mman.h>:

Flag	Effect on the New Interval
MAP_SHARED	The mapping can be shared.
MAP_PRIVATE	The mapping cannot be shared.

Flag	Effect on the New Interval
MAP_FIXED	The new interval must start at the given address addr.
MAP_ANONYMOUS	The mapping is not file-backed, but is anonymous.
MAP_GROWSDOWN	Corresponds to VM_GROWSDOWN.
MAP_DENYWRITE	Corresponds to VM_DENYWRITE.
MAP_EXECUTABLE	Corresponds to VM_EXECUTABLE.
MAP_LOCKED	Corresponds to VM_LOCKED.
MAP_NORESERVE	No need to reserve space for the mapping.
MAP_POPULATE	Populate (prefault) page tables.
MAP_NONBLOCK	Do not block on I/O.

If any of the parameters are invalid, do_mmap() returns a negative value. Otherwise, a suitable interval in virtual memory is located. If possible, the interval is merged with an adjacent memory area. Otherwise, a new vm_area_struct structure is allocated from the vm_area_cachep slab cache, and the new memory area is added to the address space's linked list and red-black tree of memory areas via the vma_link() function. Next, the total_vm field in the memory descriptor is updated. Finally, the function returns the initial address of the newly created address interval.

The do_mmap() functionality is exported to user-space via the mmap() system call, defined as:

```
void * mmap2(void *start, size_t length, int prot, int flags, int fd, off_t pgoff)
```

munmap() and do_munmap(): Removing an Address Interval

The do_munmap() function removes an address interval from a specified process address space. The function is declared in <linux/mm.h>:

```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
```

The first parameter mm specifies the address space from which the interval starting at address start of length len bytes is removed. On success, zero is returned. Otherwise, a negative error code is returned.

As the complement of the mmap(), the munmap() system call is exported to user-space as a means to enable processes to remove address intervals from their address space:

```
int munmap(void *start, size_t length)
```

The system call is defined in mm/mmap.c and acts as a simple wrapper to do_munmap():

```
asmlinkage long sys_munmap(unsigned long addr, size_t len)
```

```
{
    int ret;
    struct mm_struct *mm;
    mm = current->mm;
    down_write(&mm->mmap_sem);
    ret = do_munmap(mm, addr, len);
    up_write(&mm->mmap_sem);
    return ret;
}
```

Page Tables

In Linux, the page tables consist of three levels.

- The top-level page table is the page global directory (PGD), which consists of an array of pgd_t types. On most architectures, the pgd_t type is an unsigned long. The entries in the PGD point to entries in the second-level directory, the PMD.
- The second-level page table is the page middle directory (PMD), which is an array of pmd_t types. The entries in the PMD point to entries in the PTE.
- The final level is called simply the page table and consists of page table entries of type pte_t. Page table entries point to physical pages.

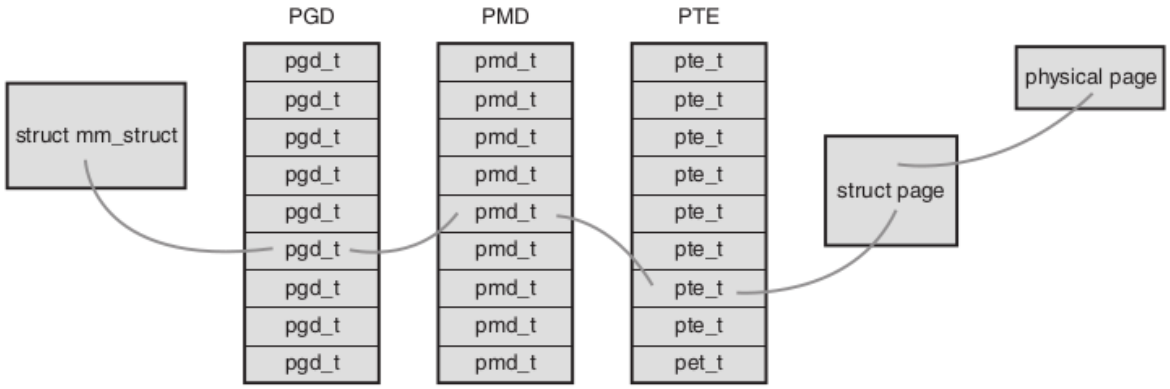

Figure 15.1 Virtual-to-physical address lookup.

Each process has its own page tables (threads share them). The pgd field of the memory descriptor points to the process's page global directory. Manipulating and traversing page tables requires the page_table_lock, which is located inside the associated memory descriptor.

Page table data structures are quite architecture-dependent and thus are defined in <asm/page.h>.

Most processors implement a translation lookaside buffer (TLB), which acts as a hardware cache of virtual-to-physical mappings. When accessing a virtual address, the processor first checks whether the mapping is cached in the TLB. If there is a hit, the physical address is immediately returned. If there is a miss, the page tables are consulted for the corresponding physical address.

16)

The Linux kernel implements a disk cache called the page cache. The goal of this cache is to minimize disk I/O by storing data in physical memory that would otherwise require disk access.

Two factors comingle to make disk caches a critical component of any modern operating system:

- disk access is several orders of magnitude slower than memory access.
- data accessed once will, with a high likelihood, find itself accessed again in the near future. This principle is called temporal locality.

Approaches to Caching

The page cache consists of physical pages in RAM, the contents of which correspond to physical blocks on a disk. The size of the page cache is dynamic. We call the storage device being cached the backing store. Whenever the kernel begins a read operation, it first checks if the requisite data is in the page cache. If it is, the kernel can forgo accessing the disk and read the data directly out of RAM. This is called a cache hit. If the data is not in the cache, called a cache miss, the kernel must schedule block I/O operations to read the data off the disk. After the data is read off the disk, the kernel populates the page cache with the data so that any subsequent reads can occur out of the cache.

Write Caching

When a process writes to disk, the strategy in Linux is called write-back. In a write-back cache, processes perform write operations directly into the page cache. The backing store is not immediately or directly updated. Instead, the written-to pages in the page cache are marked as dirty and are added to a dirty list. Periodically, pages in the dirty list are written back to disk in a process called writeback, bringing the on-disk copy in line with the in-memory cache. The pages are then marked as no longer dirty.

Cache Eviction

The strategy that decides what to remove, is called cache eviction.

Least Recently Used

One of the more successful algorithms is called least recently used, or LRU. An LRU eviction strategy requires keeping track of when each page is accessed and evicting the pages with the oldest timestamp. However, one particular failure of the LRU strategy is that many files are accessed once and then never again.

The Two-List Strategy

Linux implements a modified version of LRU, called the two-list strategy. Linux keeps two list: the active list and the inactive list. Pages on the active list are considered "hot" and are not available for eviction. Pages on the inactive list are available for cache eviction. Pages are placed on the active list only when they are accessed while already residing on the inactive list. Both list are maintained in a pseudo-LRU manner: Items are added to the tail and removed from the head. The lists are kept in balance. This two-list approach is known as LRU/2, it can be generalized to n-lists, called LRU/n.

The Linux Page Cache

The address_space Object

The Linux page cache used a new Object to manage entries in the cache and page I/O operations. That Object is the address_space structure.

```
struct address_space {
    struct inode          *host;          /* owning inode */
    struct radix_tree_root  page_tree;      /* radix tree of all pages */
    spinlock_t            tree_lock;      /* page_tree lock */
    unsigned int          i_mmap_writable;  /* VM_SHARED ma count */
    struct prio_tree_root   i_mmap;         /* list of all mappings */
    struct list_head      i_mmap_nonlinear;  /* VM_NONLINEAR ma list */
    spinlock_t            i_mmap_lock;      /* i_mmap lock */
    atomic_t              truncate_count;   /* truncate re count */
    unsigned long         nrpages;          /* total number of pages */
    pgoff_t               writeback_index;  /* writeback start offset */
    struct address_space_operations   *a_ops;  /* operations table */
    unsigned long         flags;            /* gfp_mask and error flags */
    struct backing_dev_info *backing_dev_info;  /* read-ahead information */
    spinlock_t            private_lock;     /* private lock */
    struct list_head      private_list;     /* private list */
    struct address_space    *assoc_mapping;   /* associated buffers */
};
```

address_space operations

The a_ops field points to the address space operations table, which is represented by struct address_space_operations and is defined in <linux/fs.h>:

```
struct address_space_operations {
    int (*writepage)(struct page *, struct writeback_control *);
    int (*readpage) (struct file *, struct page *);
    int (*sync_page) (struct page *);
    int (*writepages) (struct address_space *,
                struct writeback_control *);
    int (*set_page_dirty) (struct page *);
    int (*readpages) (struct file *, struct address_space *,
                struct list_head *, unsigned);
```

```
        int (*write_begin)(struct file *, struct address_space *mapping,
                        loff_t pos, unsigned len, unsigned flags,
                        struct page **pagep, void **fsdata);
        int (*write_end)(struct file *, struct address_space *mapping,
                        loff_t pos, unsigned len, unsigned copied,
                        struct page *page, void *fsdata);
        sector_t (*bmap) (struct address_space *, sector_t);
        int (*invalidatepage) (struct page *, unsigned long);
        int (*releasepage) (struct page *, int);
        int (*direct_IO) (int, struct kiocb *, const struct iovec *,
                       loff_t, unsigned long);
        int (*get_xip_mem) (struct address_space *, pgoff_t, int,
                        void **, unsigned long *);
        int (*migratepage) (struct address_space *,
                        struct page *, struct page *);
        int (*launder_page) (struct page *);
        int (*is_partially_uptodate) (struct page *,
                                read_descriptor_t *,
                                unsigned long);
        int (*error_remove_page) (struct address_space *,
                                struct page *);
    };
```

These function points at the functions that implement page I/O for this cached object. Each backing store describes how it interacts with the page cache via its own address_space_operations.
The readpage() and writepage() methods are most important.

Starting with a page read operation. First, the Linux kernel attempts to find the request data in the page cache. The find_get_page() method is used to perform this check, it is passed an address_space and page offset. These values search the page cache for the desired data:

```
page = find_get_page(mapping, index);
```

Here, mapping is the given address_space and index is the desired offset into the file, in pages. If the page does not exist in the cache, find_get_page returns NULL and a new page is allocated and added to the page cache:

```
struct page *page;
```

```
int error;


/* allocate the page ... */
page = page_cache_alloc_cold(mapping);
if (!page)
    /* error allocating memory */


/* ... and then add it to the page cache */
error = add_to_page_cache_lru(page, mapping, index, GFP_KERNEL);
if (error)
    /* error adding page to page cache */
```

Finally, the requested data can be read from disk, added to the page cache, and returned to the user:

```
error = mapping->a_ops->readpage(file, page);
```

Write operations are a bit different. For file mappings, whenever a page is modified, the VM simply calls:

```
SetPageDirty(page);
```

The kernel later writes the page out via the writepage() method. Write operations on specific files are more complicated. The generic write path in mm/filemap.c performs the following steps:

```
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
status = a_ops->prepare_write(file, page, offset, offset+bytes);
page_fault = filemap_copy_from_user(page, offset, buf, bytes);
status = a_ops->commit_write(file, page, offset, offset+bytes);
```

First, the page cache is searched for the desired page. If it is not in the cache, an entry is allocated and added. Next, the kernel sets up the write request and the data is copied from user-space into a kernel buffer. Finally, the data is written to disk.

Radix Tree

Each address_space has a unique radix tree stored as page_tree. A radix tree is a type of binary tree. The radix tree enables quick searching for the desired page, given only the file offset. Page cache searching functions such as find_get_page() call radix_tree_lookup(), which performs a search on the given tree for the given object.

The core radix tree code is available in generic form in lib/radix-tree.c. Users of the radix tree need to include <linux/radix-tree.h>.

The Buffer Cache

Individual disk blocks also tie into the page cache, by way of block I/O buffers. Buffers act as descriptors that map pages in memory to disk blocks; thus, the page cache also reduces disk access during block I/O operations by both caching disk blocks and buffering block I/O operations until later. This caching is often referred to as the buffer cache, although as implemented it is not a separate cache but is part of the page cache.

The Flusher Threads

When data in the page cache is newer than the data on the backing store, we call that data dirty. Dirty pages that accumulate in memory eventually need to be written back to disk. Dirty page writeback occurs in three situations:

- When free memory shrinks below a specified threshold, the kernel writes dirty data back to disk. When clean, the kernel can evict the data from the cache and then shrink the cache, freeing up more memory.
- When dirty data grows older than a specific threshold, sufficiently old data is written back to disk to ensure that dirty data does not remain dirty indefinitely.
- When a user process invokes the sync() and fsync() system calls, the kernel performs writeback on demand.

A gang of kernel threads, the flusher threads, performs all three jobs.

The system administrator can set these values either in /proc/sys/vm or via sysctl:

Variable	Description
dirty_background_ratio	As a percentage of total memory, the number of pages at which the flusher threads begin writeback of dirty data.
dirty_expire_interval	In milliseconds, how old data must be to be written out the next time a flusher thread wakes to perform periodic writeback.
dirty_ratio	As a percentage of total memory, the number of pages a process generates before it begins writeback of dirty data.
dirty_writeback_interval	In milliseconds, how often a flusher thread should wake up to write data back out to disk.
laptop_mode	A Boolean value controlling laptop mode. See the following section.

The flusher code lives in mm/page-writeback.c and mm/backing-dev.c and the writeback mechanism lives in fs/fs-writeback.c.

17)

- Device types: Classifications used in all Unix systems to unify behaviour of common devices
- Modules: The mechanism by which the Linux kernel can load and unload object code on demand

- Kernel objects: Support for adding simple object-oriented behaviour and a parent/child relationship to kernel data structures
- Sysfs: A filesystem representation of the system's device tree

Device Types

In Linux, as with all Unix systems, devices are classified into one of three types:

- Block devices: Often abbreviated blkdevs, block devices are addressable in device-specified chunks called blocks and generally support seeking, the random access of data.
- Character devices: Often abbreviated cdevs, character devices are generally not addressable, providing access to data only as a stream, generally of characters (bytes).
- Network devices: Sometimes called Ethernet devices after the most common type of network devices, network devices provide access to a network via a physical adapter and a specific protocol.

Modules

The Linux kernel is modular, supporting the dynamic insertion and removal of code from itself at runtime. Related subroutines, data, and entry and exit points are grouped together in a single binary image, a loadable kernel object, called a module.

The Device Model

The device model provides a single mechanism for representing devices and describing their topology in the system.

sysfs

The sysfs filesystem is an in-memory virtual filesystem that provides a view of the kobject hierarchy. It enables users to view the device topology of their system as a simple filesystem. Using attributes, kobjects can export files that enable kernel variables to be read from and optionally written to.

The magic behind sysfs is simply tying kobjects to directory entries via the dentry member inside each kobject.