Chapter 5. Kernel Synchronization

- You could think of the kernel as a **server that answers requests**; these requests can come either **from a process** running on a CPU **or an external device** issuing an interrupt request.

- We make this analogy to underscore that **parts of the kernel are not run serially**, but in an **interleaved way**. Thus, they can give rise to race conditions, which must be controlled through proper **synchronization techniques**.

- A general introduction to these topics can be found in the section "An Overview of Unix Kernels" in Chapter 1 of the book.

- We start this chapter by reviewing when, and to what extent, kernel requests are executed in an interleaved fashion.

- We then introduce the basic synchronization primitives implemented by the kernel and describe how they are applied in the most common cases.

5.1.1. Kernel Preemption

- a kernel is preemptive if a **process switch** may occur while the replaced process is executing a kernel function, that is, while it runs **in Kernel Mode**.
- Unfortunately, in Linux (as well as in any other real operating system) things are much more complicated:
  - Both in preemptive and nonpreemptive kernels, a process running in Kernel Mode can voluntarily **relinquish the CPU**, for instance because it has to sleep waiting for some resource. We will call this kind of process switch a **planned process switch**.
  - However, a preemptive kernel differs from a nonpreemptive kernel on the way a process running in Kernel Mode **reacts to asynchronous events** that could induce a process switch - for instance, an interrupt handler that awakes a higher priority process. We will call this kind of process switch a **forced process switch**.
  - All process switches are performed by the **switch_to macro**. In both preemptive and nonpreemptive kernels, a process switch occurs when a process has finished some thread of kernel activity and the scheduler is invoked. However, **in nonpreemptive kernels, the current process cannot be replaced unless it is about to switch to User Mode.**
- Therefore, the main characteristic of a **preemptive kernel** is that a process running in Kernel Mode can be replaced by another process while **in the middle of a kernel function.**

examples to illustrate the difference between preemptive and nonpreemptive kernels.

- While **process A executes an exception handler, a higher priority process B becomes runnable**. This could happen, for instance, if an IRQ occurs and the corresponding handler awakens process B.
  - If the kernel is preemptive, a forced process switch replaces process A with B. The exception handler is left unfinished and will be resumed only when the scheduler selects again process A for execution.
  - Conversely, if the kernel is nonpreemptive, no process switch occurs until process A either finishes handling the exception handler or voluntarily relinquishes the CPU.
- For another example, consider a process that executes an exception handler and whose time quantum expires.
  - If the kernel is preemptive, the process may be replaced immediately;
  - however, if the kernel is nonpreemptive, the process continues to run until it finishes handling the exception handler or voluntarily relinquishes the CPU.

Dispatch latency and disabling preemption

- The main motivation for making a kernel preemptive is to **reduce the dispatch latency of the User Mode processes**, that is, the delay between the time they become runnable and the time they actually begin running.
- Processes performing timely scheduled tasks (such as external hardware controllers, environmental monitors, movie players, and so on) really benefit from kernel preemption, because it reduces the risk of being delayed by another process running in Kernel Mode.
- Preemption is **disabled** when:
  - The kernel is executing an **interrupt service routine**.
  - The **deferrable functions are disabled** (always true when the kernel is executing a softirq or tasklet).
  - The kernel preemption has been **explicitly disabled** by setting the preemption counter to a positive value.
- The above rules tell us that the **kernel can be preempted only when it is executing an exception handler (in particular a system call)** and the kernel preemption has not been explicitly disabled.
- Furthermore the **local CPU must have local interrupts enabled**, otherwise kernel preemption is not performed.

Table 5-1 deal with the preemption counter in the prempt_count field.

5.1.2. When Synchronization Is Necessary

- Chapter 1 introduced the concepts of **race condition and critical region** for processes.
- The **same definitions apply to kernel control paths** . In this chapter, a race condition can occur when the outcome of a computation depends on how two or more interleaved kernel control paths are nested.
- A **critical region** is a section of code that must be completely executed by the kernel control path that enters it before another kernel control path can enter it.
- Interleaving kernel control paths complicates the life of kernel developers: they must apply special care in order to **identify the critical regions in exception handlers, interrupt handlers, deferrable functions, and kernel threads .**
- Once a critical region has been identified, it must be **suitably protected** to ensure that any time at most one kernel control path is inside that region.

examples

- Suppose, for instance, that **two different interrupt handlers** need to access the **same data structure** that contains several related member variables for instance, a buffer and an integer indicating its length.
  - All statements affecting the data structure must be put into a single **critical region**. If the system includes a **single CPU**, the critical region can be implemented by **disabling interrupts** while accessing the shared data structure, because nesting of kernel control paths can only occur when interrupts are enabled.
- On the other hand, if the same data structure is **accessed only by the service routines of system calls**, and if the system includes a **single CPU**, the critical region can be implemented quite simply by **disabling kernel preemption** while accessing the shared data structure.
- As you would expect, **things are more complicated in multiprocessor systems**. Many CPUs may execute kernel code at the same time, so kernel developers cannot assume that a data structure can be safely accessed just because kernel preemption is disabled and the data structure is never addressed by an interrupt, exception, or softirq handler.
- We'll see in the following sections that **the kernel offers a wide range of different synchronization techniques**. It is up to kernel designers to solve each synchronization problem by selecting the most efficient technique.

5.1.3. When Synchronization Is Not Necessary

- **Some design choices already discussed in the previous chapter simplify somewhat the synchronization of kernel control paths. Let us recall them briefly:**
  - All interrupt handlers acknowledge the interrupt on the PIC and also disable the IRQ line. **Further occurrences** of the **same interrupt** cannot occur **until the handler terminates**.
  - **Interrupt handlers, softirqs, and tasklets are both nonpreemptable and non-blocking**, so they cannot be suspended for a long time interval.
    - In the worst case, their execution will be slightly delayed, because **other interrupts occur during their execution** (nested execution of kernel control paths).
  - A kernel control path performing **interrupt handling cannot be interrupted** by a kernel control path executing a **deferrable function or a system call** service routine.
  - **Softirqs and tasklets** cannot be interleaved on a given CPU.
  - The same **tasklet** cannot be executed simultaneously on several CPUs.

Code simplification

- Each of the above design choices can be viewed as a constraint that can be exploited to code some kernel functions more easily.
- Here are a few **examples of possible simplifications:**
  - **Interrupt handlers and tasklets** need not to be coded as reentrant functions.
  - **Per-CPU variables accessed by softirqs and tasklets** only do not require synchronization.
  - A **data structure accessed by only one kind of tasklet** does not require synchronization.
- The rest of this chapter describes what to do when synchronization is necessary i.e., how to prevent data corruption due to unsafe accesses to shared data structures.

5.2. Synchronization Primitives

- We now examine how kernel control paths can be **interleaved while avoiding race conditions** among shared data.
- Table 5-2 lists the synchronization techniques used by the Linux kernel.
- The "**Scope**" column indicates whether the synchronization technique applies to all CPUs in the system or to a single CPU.
  - For instance, **local interrupt disabling** applies to just **one CPU** (other CPUs in the system are not affected);
  - conversely, an **atomic operation** affects **all CPUs** in the system (atomic operations on several CPUs cannot interleave while accessing the same data structure).
- Let's now briefly discuss each synchronization technique.

Table 5-2. Various types of synchronization techniques used by the kernel

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update	Lock-free access to shared data structures through pointers	All CPUs

5.2.1. Per-CPU Variables

- The best synchronization technique consists in designing the kernel so as to avoid the need for synchronization in the first place. As we'll see, in fact, every **explicit synchronization primitive has a significant performance cost**.

- The simplest and most efficient synchronization technique consists of declaring kernel variables as per-CPU variables . Basically, **a per-CPU variable is an array of data structures, one element per each CPU in the system.**

- A CPU should not access the elements of the array corresponding to the other CPUs; on the other hand, it can freely **read and modify its own element without fear of race conditions**, because it is the only CPU entitled to do so.

- This also means, however, that the per-CPU variables can be used only in particular cases, basically, when it **makes sense to logically split the data across the CPUs** of the system.

- The elements of the per-CPU array are **aligned in main memory** so that each data structure falls on a **different line of the hardware cache**. Therefore, concurrent accesses to the per-CPU array do not result in **cache line snooping and invalidation**, which are costly operations in terms of system performance (“**false sharing**”).

per-CPU

- While per-CPU variables provide protection against concurrent accesses from several CPUs, they **do not provide protection against accesses from asynchronous functions** (interrupt handlers and deferrable functions).
- In these cases, **additional synchronization primitives are required.**
- Furthermore, per-CPU variables are **prone to race conditions caused by kernel preemption** , both in uniprocessor and multiprocessor systems.
- As a general rule, a kernel control path should access a per-CPU variable with **kernel preemption disabled**.
  - Just consider, for instance, what would happen if a kernel control path gets the address of its local copy of a per-CPU variable, and then it is **preempted and moved to another CPU**: the address still refers to the element of the previous CPU.

5.2.2. Atomic Operations

- Several assembly language instructions are of type "**read-modify-write**" that is, they **access a memory location twice**, the first time to read the old value and the second time to write a new value.
- Suppose that two kernel control paths running on two CPUs try to "read-modify-write" the same memory location at the same time by executing nonatomic operations.
  - At first, both CPUs try to read the same location, but the memory arbiter (a hardware circuit that serializes accesses to the RAM chips) steps in to grant access to one of them and delay the other.
  - However, when the first read operation has completed, the delayed CPU reads exactly the same (old) value from the memory location.
  - Both CPUs then try to write the same (new) value to the memory location; again, the bus memory access is serialized by the memory arbiter, and eventually both write operations succeed.
  - However, the global result is incorrect because both CPUs write the same (new) value. Thus, the two interleaving "read-modify-write" operations act as a single one.
- The easiest way to prevent race conditions due to "read-modify-write" instructions is by **ensuring that such operations are atomic at the chip level**. Every such operation must be executed in a single instruction without being interrupted in the middle and avoiding accesses to the same memory location by other CPUs. These very small atomic operations can be found at the base of other, more flexible mechanisms to create critical regions.

80x86 Instructions and kernel atomic type and operations

- Assembly language instructions that make **zero or one aligned memory** access are **atomic**.
- **Read-modify-write** assembly language instructions (such as inc or dec) that read data from memory, update it, and write the updated value back to memory are **atomic** if no other processor has taken the memory bus after the read and before the write. Memory bus stealing never happens in a **uniprocessor system**.
- **Read-modify-write** assembly language instructions whose **opcode is prefixed by the lock byte** (0xf0) are atomic even on a multiprocessor system. When the control unit detects the prefix, it "locks" the memory bus until the instruction is finished. Therefore, other processors cannot access the memory location while the locked instruction is being executed.
- Assembly language instructions whose **opcode is prefixed by a rep byte** (0xf2, 0xf3, which forces the control unit to repeat the same instruction several times) are **not atomic**. The control unit **checks for pending interrupts before executing a new iteration**.
- When you write C code, you cannot guarantee that the compiler will use an atomic instruction for an operation like a=a+1 or even for a++. Thus, the Linux kernel provides a special **atomic_t type** (an atomically accessible counter) and some special functions and macros that act on atomic_t variables and are implemented as single, atomic assembly language instructions. On **multiprocessor systems, each such instruction is prefixed by a lock byte.**

5.2.3. Optimization and Memory Barriers

- When using **optimizing compilers**, you should never take for granted that instructions will be performed in the **exact order** in which they appear in the source code.
  - For example, a compiler might reorder the assembly language instructions in such a way to optimize how registers are used.
- Moreover, **modern CPUs** usually execute several **instructions in parallel and might reorder memory accesses**. These kinds of reordering can greatly speed up the program.
- When dealing with synchronization, however, reordering instructions must be avoided.
  - Things would quickly become hairy if an instruction placed after a synchronization primitive is executed before the synchronization primitive itself.
- **Therefore, all synchronization primitives act as optimization and memory barriers**.

Optimization barriers

- An optimization barrier primitive ensures that the assembly language instructions corresponding to C statements placed before the primitive are not mixed by the compiler with assembly language instructions corresponding to C statements placed after the primitive.
- In Linux the barrier( ) macro, which expands into **asm volatile("":::"memory")**, acts as an optimization barrier.
- The **asm** instruction tells the compiler to insert an assembly language fragment (empty, in this case).
- The **volatile** keyword forbids the compiler to reshuffle the asm instruction with the other instructions of the program.
- The **memory** keyword forces the compiler to assume that all memory locations in RAM have been changed by the assembly language instruction; therefore, the compiler cannot optimize the code by using the values of memory locations stored in CPU registers before the asm instruction.
- Notice that the **optimization barrier does not ensure that the executions of the assembly language instructions are not mixed by the CPU**, this is a job for a memory barrier.

Memory barriers

- A memory barrier primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive.
- Thus, a memory barrier is like a firewall that cannot be passed by an assembly language instruction.
- In the **80x86 processors**, the following kinds of assembly language **instructions** are said to be **"serializing"** because they act as memory barriers:
  - All instructions that operate on **I/O ports**
  - All instructions **prefixed by the lock byte**
  - All instructions that **write into control registers, system registers, or debug registers** (for instance, cli and sti)
  - The **lfence , sfence , and mfence assembly language instructions**, which have been introduced in the Pentium 4 microprocessor to efficiently implement read memory barriers, write memory barriers, and read-write memory barriers, respectively.
  - A few **special assembly language instructions**; among them, the iret instruction that terminates an interrupt or exception handler

- Linux uses a few memory barrier primitives, which are shown in Table 5-6. These primitives act also as optimization barriers, because we must make sure the compiler does not move the assembly language instructions around the barrier. "Read memory barriers" act only on instructions that read from memory, while "write memory barriers" act only on instructions that write to memory. Memory barriers can be useful in both multiprocessor and uniprocessor systems. The smp_xxx( ) primitives are used whenever the memory barrier should prevent race conditions that might occur only in multiprocessor systems; in uniprocessor systems, they do nothing. The other memory barriers are used to prevent race conditions occurring both in uniprocessor and multiprocessor systems.

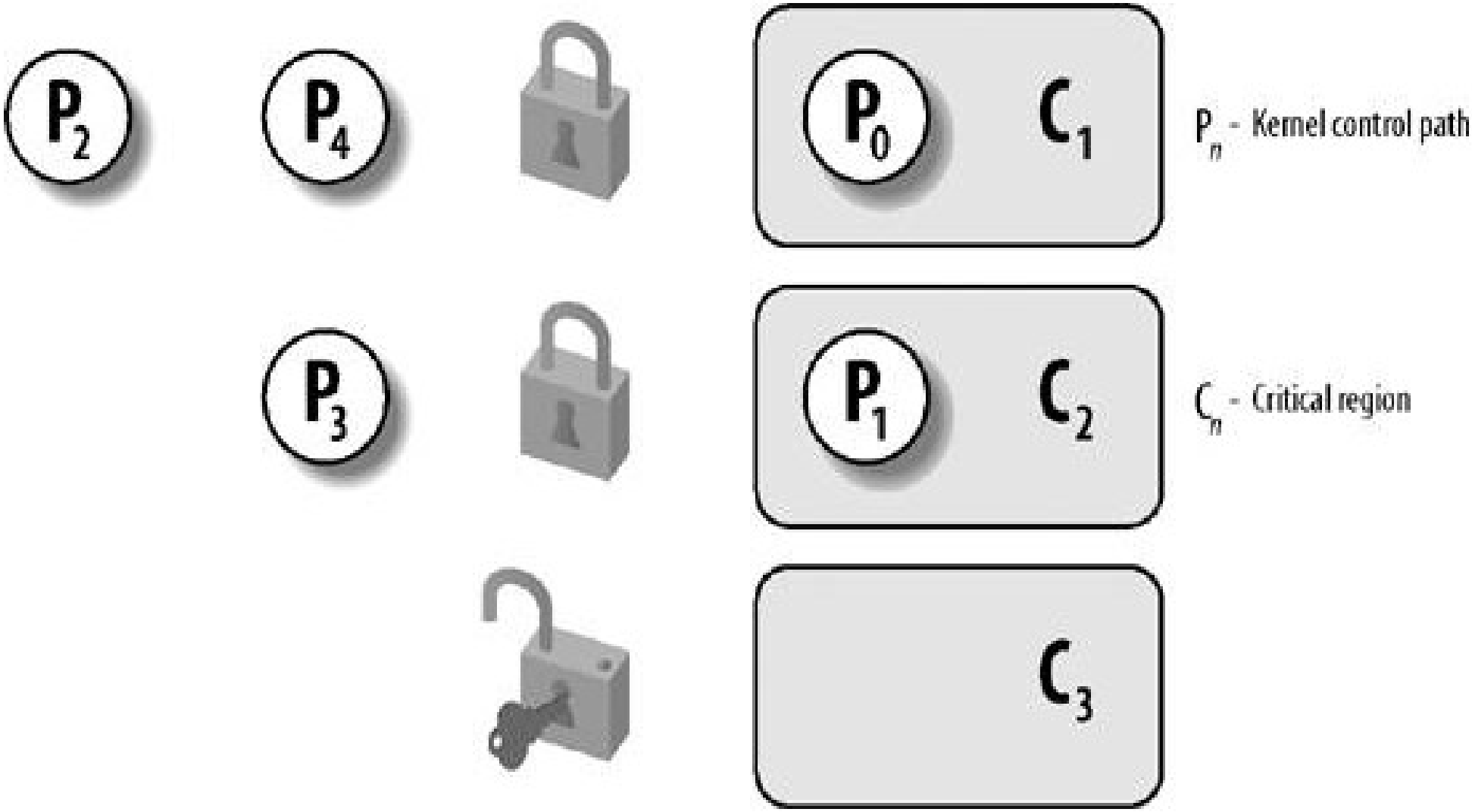
Implementations

- The implementations of the memory barrier primitives depend on the architecture of the system.
  - On an 80x86 microprocessor, the **rmb( )** macro usually expands into asm volatile("lfence") if the CPU supports the lfence assembly language instruction, or into asm volatile("lock;addl $0,0(%%esp)":::"memory") otherwise.
    - The lock; addl $0,0(%%esp) assembly language instruction adds zero to the memory location on top of the stack; the instruction is useless by itself, but the lock prefix makes the instruction a memory barrier for the CPU.
  - The **wmb( )** macro is actually simpler because it expands into barrier( ). This is because existing **Intel microprocessors never reorder write memory accesses**, so there is no need to insert a serializing assembly language instruction in the code. The macro, however, **forbids the compiler from shuffling the instructions**.
- Notice that **in multiprocessor systems, all atomic operations** described in the earlier section "Atomic Operations" **act as memory barriers** because they use the lock byte.

5.2.4. Spin Locks

- When a kernel control path must access a shared data structure or enter a critical region, it needs to acquire a "lock" for it.
- A resource protected by a locking mechanism is quite similar to a resource confined in a room whose door is locked when someone is inside.
- If a kernel control path wishes to access the resource, it tries to "open the door" by acquiring the lock. It succeeds only if the resource is free. Then, as long as it wants to use the resource, the door remains locked. When the kernel control path releases the lock, the door is unlocked and another kernel control path may enter the room.
- Figure 5-1 illustrates the use of locks. Five kernel control paths (P0, P1, P2, P3, and P4) are trying to access two critical regions (C1 and C2). Kernel control path P0 is inside C1, while P2 and P4 are waiting to enter it. At the same time, P1 is inside C2, while P3 is waiting to enter it. Notice that P0 and P1 could run concurrently. The lock for critical region C3 is open because no kernel control path needs to enter it.

Figure 5-1. Protecting critical regions with several locks



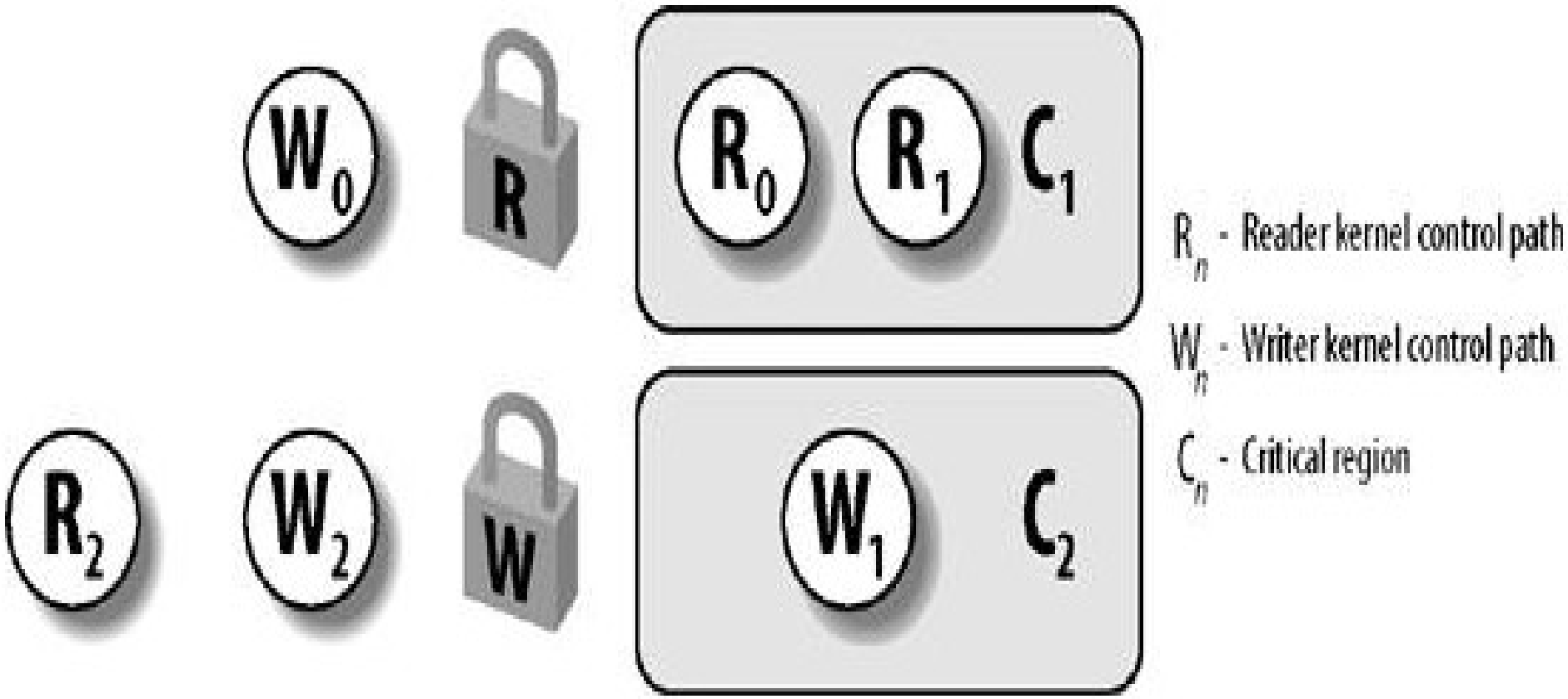
Spin locks are a special kind of lock designed to work in a multiprocessor environment.

- If the kernel control path finds the spin lock "open," it acquires the lock and continues its execution.
- Conversely, if the kernel control path finds the lock "closed" by a kernel control path running on **another CPU**, it "spins" around, repeatedly executing a **tight instruction loop**, until the lock is released.
- The instruction loop of spin locks represents a "**busy wait**." The waiting kernel control path **keeps running on the CPU**, even if it has nothing to do besides waste time.
- Nevertheless, spin locks are usually **convenient**, because many kernel resources are locked for a **fraction of a millisecond only**; therefore, it would be far more time-consuming to release the CPU and reacquire it later.
- As a general rule, **kernel preemption is disabled in every critical region protected by spin locks**.
- In the case of a **uniprocessor system**, the locks themselves are **useless**, and the spin lock primitives just disable or enable the kernel preemption.
- Please notice that kernel preemption is still enabled during the busy wait phase, thus a process waiting for a spin lock to be released could be replaced by a higher priority process.

5.2.5. Read/Write Spin Locks

- Read/write spin locks have been introduced to **increase the amount of concurrency** inside the kernel.
- They allow **several kernel control paths to simultaneously read the same data structure**, as long as no kernel control path modifies it.
- If a kernel control path wishes to write to the structure, it must acquire the **write version** of the read/write lock, which grants **exclusive access** to the resource.
- Of course, allowing concurrent reads on data structures **improves system performance**.
- Figure 5-2 illustrates two critical regions (C1 and C2) protected by read/write locks. Kernel control paths R0 and R1 are reading the data structures in C1 at the same time, while W0 is waiting to acquire the lock for writing. Kernel control path W1 is writing the data structures in C2, while both R2 and W2 are waiting to acquire the lock for reading and writing, respectively.

Figure 5-2. Read/write spin locks



5.2.6. Seqlocks

- When using read/write spin locks, requests issued by kernel control paths to perform a read_lock or a write_lock operation have the **same priority**: readers must wait until the writer has finished and, similarly, a writer must wait until all readers have finished.
- Seqlocks introduced in Linux 2.6 are similar to read/write spin locks, except that they give a much **higher priority to writers**:
  - in fact a **writer is allowed to proceed even when readers are active**. The good part of this strategy is that a **writer never waits (unless another writer is active)**;
  - the bad part is that a **reader may sometimes be forced to read the same data several times** until it gets a valid copy.
- Each seqlock is a seqlock_t structure consisting of two fields:
  - a lock field of type spinlock_t and an integer sequence field.
  - This second field plays the role of a **sequence counter**. Each **reader** must read this sequence counter **twice**, **before and after reading the data**, and check whether the **two values coincide**.
  - In the opposite case, a new writer has become active and has increased the sequence counter, thus implicitly telling the reader that the data just read is not valid.

5.2.7. Read-Copy Update (RCU)

- Read-copy update (RCU) is yet another synchronization technique designed to protect data structures that are mostly accessed for reading by several CPUs. RCU allows many readers and many writers to proceed concurrently (an improvement over seqlocks, which allow only one writer to proceed). Moreover, RCU is lock-free, that is, it uses no lock or counter shared by all CPUs; this is a great advantage over read/write spin locks and seqlocks, which have a high overhead due to cache line-snooping and invalidation.
- How does RCU obtain the surprising result of synchronizing several CPUs without shared data structures? The key idea consists of limiting the scope of RCU as follows:Only data structures that are dynamically allocated and referenced by means of pointers can be protected by RCU.
- No kernel control path can sleep inside a critical region protected by RCU.
- When a kernel control path wants to read an RCU-protected data structure, it executes the rcu_read_lock( ) macro, which is equivalent to preempt_disable( ). Next, the reader dereferences the pointer to the data structure and starts reading it. As stated above, the reader cannot sleep until it finishes reading the data structure; the end of the critical region is marked by the rcu_read_unlock( ) macro, which is equivalent to preempt_enable( ).
- Because the reader does very little to prevent race conditions, we could expect that the writer has to work a bit more. In fact, when a writer wants to update the data structure, it dereferences the pointer and makes a copy of the whole data structure. Next, the writer modifies the copy. Once finished, the writer changes the pointer to the data structure so as to make it point to the updated copy. Because changing the value of the pointer is an atomic operation, each reader or writer sees either the old copy or the new one: no corruption in the

5.2.8. Semaphores

- Essentially, they implement a locking primitive that allows waiters to sleep until the desired resource becomes free.
- Actually, Linux offers two kinds of semaphores:
  - Kernel semaphores, which are used by kernel control paths
  - System V IPC semaphores, which are used by User Mode processes
- In this section, we focus on kernel semaphores
- A kernel semaphore is similar to a spin lock, in that it doesn't allow a kernel control path to proceed unless the lock is open.
- However, whenever a kernel control path **tries to acquire a busy resource** protected by a kernel semaphore, the corresponding process **is suspended**. It becomes runnable again when the resource is released.
- Therefore, kernel semaphores can be **acquired only by functions that are allowed to sleep:**
  - **interrupt handlers and deferrable functions cannot use them.**

5.2.9. Read/Write Semaphores

- Read/write semaphores are similar to the read/write spin locks described earlier in the section "Read/Write Spin Locks," except that waiting processes are suspended instead of spinning until the semaphore becomes open again.
- Many kernel control paths may concurrently acquire a read/write semaphore for reading; however, every writer kernel control path must have exclusive access to the protected resource. Therefore, the semaphore can be acquired for writing only if no other kernel control path is holding it for either read or write access.
- Read/write semaphores improve the amount of concurrency inside the kernel and improve overall system performance.
- The kernel handles all processes waiting for a read/write semaphore in **strict FIFO** order. Each reader or writer that finds the semaphore closed is inserted in the last position of a semaphore's wait queue list. When the semaphore is released, the process in the first position of the wait queue list are checked. The first process is always awoken. If it is a writer, the other processes in the wait queue continue to sleep. If it is a reader, all readers at the start of the queue, up to the first writer, are also woken up and get the lock. However, readers that have been queued after a writer continue to sleep.

5.2.11. Local Interrupt Disabling

- Interrupt disabling is one of the **key mechanisms** used to ensure that a sequence of kernel statements is treated as a critical section.
- It allows a kernel control path to **continue executing even when hardware devices issue IRQ signals**, thus providing an effective way to protect data structures that are also accessed by interrupt handlers.
- By itself, however, local interrupt disabling **does not protect against concurrent accesses to data structures by interrupt handlers running on other CPUs**,
  - **so in multiprocessor systems, local interrupt disabling is often coupled with spin locks .**
- The local_irq_disable( ) macro, which makes use of the **cli** assembly language instruction, disables interrupts on the local CPU. The local_irq_enable( ) macro, which makes use of the of the **sti** assembly language instruction, enables them.
- the cli and sti assembly language instructions, respectively, **clear and set the IF flag of the eflags control register**. The irqs_disabled( ) macro yields the value one if the IF flag of the eflags register is clear, the value one if the flag is set.

- When the kernel enters a critical section, it disables interrupts by clearing the IF flag of the eflags register. But at the end of the critical section, often the kernel can't simply set the flag again. Interrupts can execute in nested fashion, so the kernel does not necessarily know what the IF flag was before the current control path executed. In these cases, the control path must save the old setting of the flag and restore that setting at the end.
- Saving and restoring the eflags content is achieved by means of the local_irq_save and local_irq_restore macros, respectively. The local_irq_save macro copies the content of the eflags register into a local variable; the IF flag is then cleared by a cli assembly language instruction. At the end of the critical region, the macro local_irq_restore restores the original content of eflags; therefore, interrupts are enabled only if they were enabled before this control path issued the cli assembly language instruction.

5.2.12. Disabling and Enabling Deferrable Functions

- that deferrable functions can be executed at unpredictable times (essentially, on termination of hardware interrupt handlers). Therefore, data structures accessed by deferrable functions must be protected against race conditions.

- the kernel sometimes needs to disable deferrable functions without disabling interrupts. Local deferrable functions can be enabled or disabled on the local CPU by acting on the softirq counter stored in the preempt_count field of the current's thread_info descriptor.