Kernel Synchronization

Introduction

- 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.
- 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.

Kernel Control Path

- Kernel functions are executed following a request that may be issued in two possible ways:
  - A process executing in User Mode causes an exception, for instance by executing an int 0x80 assembly language instruction.
  - An external device sends a signal to a Programmable Interrupt Controller by using an IRQ line, and the corresponding interrupt is enabled.
- The sequence of instructions executed in Kernel Mode to handle a kernel request is denoted as **kernel control path.**
- When a User Mode process issues a system call request, for instance, the first instructions of the corresponding kernel control path are those included in the initial part of the **system_call( )** function, while the last instructions are those included in the **ret_from_sys_call( )** function.
- A kernel control path was defined as a sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.
- Kernel control paths play a role similar to that of processes, first, no descriptor of any kind is attached to them; second, they are not scheduled through a single function, but rather by inserting sequences of instructions that stop or resume the paths into the kernel code.

- In the simplest cases, the CPU executes a kernel control path sequentially from the first instruction to the last. When one of the following events occurs, however, the CPU interleaves kernel control paths:
  - A context switch occurs. A context switch can occur only when the schedule( ) function is invoked.
  - An interrupt occurs while the CPU is running a kernel control path with interrupts enabled. In this case, the first kernel control path is left unfinished and the CPU starts processing another kernel control path to handle the interrupt.
- Interleaving kernel control path improves the throughput of programmable interrupt controllers and device controllers.
- While interleaving kernel control paths, special care must be applied to data structures that contain several related member variables, for instance, a buffer and an integer indicating its length.
- All statements affecting such a data structure must be put into a single critical section, otherwise, it is in danger of being corrupted.

Synchronization techniques

- In this chapter, a race condition can occur when the outcome of some computation depends on how two or more interleaved kernel control paths are nested.
- A critical region is any section of code that should be completely executed by each kernel control path that begins it, before another kernel control path can enter it.
- How kernel control paths can be interleaved while avoiding race conditions among shared data.
- We'll distinguish four broad types of synchronization techniques:
    - Nonpreemptability of processes in Kernel Mode
    - Atomic operations
    - Interrupt disabling
    - Locking

Non preemptability of Processes in Kernel mode

- The Linux kernel is not preemptive, that is, a running process cannot be preempted (replaced by a higher-priority process) while it remains in Kernel Mode.
- In particular, the following assertions always hold in Linux:
    - No process running in Kernel Mode may be replaced by another process, except when the former voluntarily relinquishes control of the CPU.
    - Interrupt or exception handling can interrupt a process running in Kernel Mode; however, when the interrupt handler terminates, the kernel control path of the process is resumed.
    - A kernel control path performing interrupt or exception handling can be interrupted only by another control path performing interrupt or exception handling.
- Kernel control paths dealing with nonblocking system calls are atomic with respect to other control paths started by system calls. This simplifies the implementation of many kernel functions: any kernel data structures that are not updated by interrupt or exception handlers can be safely accessed.
- However, if a process in Kernel Mode voluntarily relinquishes the CPU, it must ensure that all data structures are left in a consistent state.
- Moreover, when it resumes its execution, it must recheck the value of all previously accessed data structures that could be changed. The change could be caused by a different kernel control path, possibly running the same code on behalf of a separate

Atomic Operations

- The easiest way to prevent race conditions is by ensuring that an operation is atomic at the chip level: the operation must be executed in a single instruction.
- These very small atomic operations can be found at the base of other, more flexible mechanisms to create critical sections.
- Thus, an atomic operation is something that can be performed by executing a single assembly language instruction in an "atomic" way, that is, without being interrupted in the middle.
- Thus, the Linux kernel provides special functions that it implements as single, atomic assembly language instructions; on multiprocessor systems each such instruction is prefixed by a lock byte.

- Let's review Intel 80x86 instructions according to that classification:
    - Assembly language instructions that make zero or one 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, naturally, never happens in a uniprocessor system, because all memory accesses are made by the same processor.
    - 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.

Function	Description
atomic_read(v)	Return *v
atomic_set(v,i)	Set *v to i.
atomic_add(i,v)	Add i to *v.
atomic_sub(i,v)	Subtract i from *v.
atomic_inc(v)	Add 1 to *v.
atomic_dec(v)	Subtract 1 from *v.
atomic_dec_and_test(v)	Subtract 1 from *v and return 1 if the result is non-null, otherwise.
atomic_inc_and_test_greater_zero(v)	Add 1 to *v and return 1 if the result is positive, otherwise.
atomic_clear_mask(mask,addr)	Clear all bits of addr specified by mask.
atomic_set_mask(mask,addr)	Set all bits of addr specified by mask.

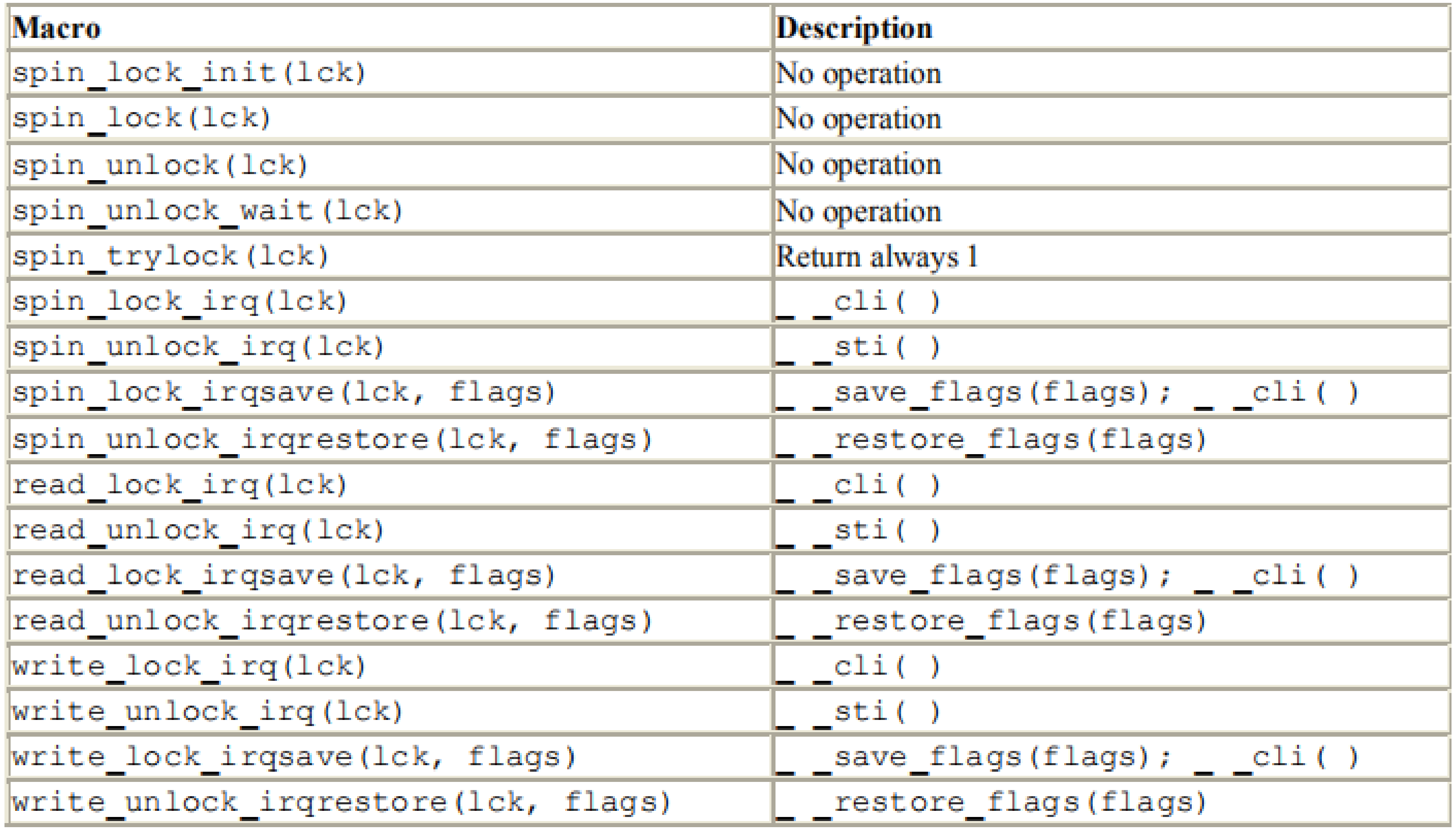
Interrupt Disabling

- Interrupt disabling is one of the key mechanisms used to ensure that a sequence of kernel statements is operated 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.
- However, interrupt disabling alone does not always prevent kernel control path interleaving.
- Indeed, a kernel control path could raise a "Page fault" exception, which in turn could suspend the current process (and thus the corresponding kernel control path). Or again, a kernel control path could directly invoke the schedule( ) function.
- This happens during most I/O disk operations because they are potentially blocking, that is, they may force the process to sleep until the I/O operation completes.
- Therefore, the kernel must never execute a blocking operation when interrupts are disabled, since the system could freeze.

- Interrupts can be disabled by means of the **cli** assembly language instruction, which is yielded by the _ _**cli( )** and **cli( )** macros. Interrupts can be enabled by means of the **sti** assembly language instruction, which is yielded by the __**sti( )** and **sti( )** macros.
- When the kernel enters a critical section, it clears the IF flag of the eflags register in order to disable interrupts. But at the end of the critical section, the kernel can't simply set the flag again.
- Interrupts can execute in nested fashion, so the kernel does not know what the IF flag was before the current control path executed.
- Each control path must therefore save the old setting of the flag and restore that setting at the end.
- In order to save the eflags content, the kernel uses the __**save_flags** macro; on a uniprocessor system it is identical to the **save_flags** macro.
- In order to restore the eflags content, the kernel uses the _ _**restore_flags** and **restore_flags** macros.

```
__save_flags(old);
__cli( );
[...]
__restore_flags(old);
```

- The __**save_flags** macro copies the content of the eflags register into the old local variable; the IF flag is then cleared by __**cli( ).**
- At the end of the critical region, the __**restore_flags** macro restores the original content of eflags; therefore, interrupts are enabled only if they were enabled before this control path issued the __**cli( )** macro.



Macro	Description
spin_lock_init(lck)	No operation
spin_lock(lck)	No operation
spin_unlock(lck)	No operation
spin_unlock_wait(lck)	No operation
spin_trylock(lck)	Return always 1
spin_lock_irq(lck)	_ _cli( )
spin_unlock_irq(lck)	_ _sti( )
spin_lock_irqsave(lck, flags)	_ _save_flags(flags); _ _cli( )
spin_unlock_irqrestore(lck, flags)	_ _restore_flags(flags)
read_lock_irq(lck)	_ _cli( )
read_unlock_irq(lck)	_ _sti( )
read_lock_irqsave(lck, flags)	_ _save_flags(flags); _ _cli( )
read_unlock_irqrestore(lck, flags)	_ _restore_flags(flags)
write_lock_irq(lck)	_ _cli( )
write_unlock_irq(lck)	_ _sti( )
write_lock_irqsave(lck, flags)	_ _save_flags(flags); _ _cli( )
write_unlock_irqrestore(lck, flags)	_ _restore_flags(flags)

- Let us recall a few examples of how these macros are used in functions:
  - The **add_wait_queue( )** and **remove_wait_queue( )** functions protect the wait queue list with the **write_lock_irqsave( )** and **write_unlock_irqrestore( )** functions.
  - The **setup_x86_irq( )** adds a new interrupt handler for a specific IRQ; the **spin_lock_irqsave( )** and **spin_unlock_irqrestore( )** functions are used to protect the corresponding list of handlers.
  - The **run_timer_list( )** function protects the dynamic timer data structures with the **spin_lock_irq( )** and **spin_unlock_irq( )** functions.
  - The **handle_signal( )** function protects the blocked field of current with the **spin_lock_irq( )** and **spin_unlock_irq( )** functions.
- Interrupt disabling is widely used by kernel functions for implementing critical regions.
- Clearly, the critical regions obtained by interrupt disabling must be short, because any kind of communication between the I/O device controllers and the CPU is blocked when the kernel enters one.
- Longer critical regions should be implemented by means of locking.

Locking Through Kernel Semaphores

- A widely used synchronization technique is locking: when a kernel control path must access a shared data structure or enter a critical region, it must 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 will succeed 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.
- Linux offers two kinds of locking: kernel semaphores, which are widely used both on uniprocessor systems and multiprocessor ones, and spin locks, which are used only on multiprocessors systems.
- When a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It will become runnable again when the resource is released.

- Kernel semaphores are objects of type struct semaphore and have these fields:
- count - Stores an integer value. If it is greater than 0, the resource is free, that is, it is currently available. Conversely, if count is less than or equal to 0, the semaphore is busy, that is, the protected resource is currently unavailable. In the latter case, the absolute value of count denotes the number of kernel control paths waiting for the resource. Zero means that a kernel control path is using the resource but no other kernel control path is waiting for it.
- wait - Stores the address of a wait queue list that includes all sleeping processes that are currently waiting for the resource. Of course, if count is greater than or equal to 0, the wait queue is empty.
- waking - Ensures that, when the resource is freed and the sleeping processes is woken up, only one of them succeeds in acquiring the resource.
- The count field is decremented when a process tries to acquire the lock and incremented when a process releases it. The MUTEX and MUTEX_LOCKED macros may be used to initialize a semaphore for exclusive access: they set the count field, respectively, to 1 (free resource with exclusive access) and (busy resource with exclusive access currently granted to the process that initializes the semaphore). Note that a semaphore could also be initialized with an arbitrary positive value n for count: in this case, at most n processes will be allowed to concurrently access the resource.
- When a process wishes to acquire a kernel semaphore lock, it invokes the down( ) function. The implementation of down( ) is quite involved, but it is essentially equivalent to the following:

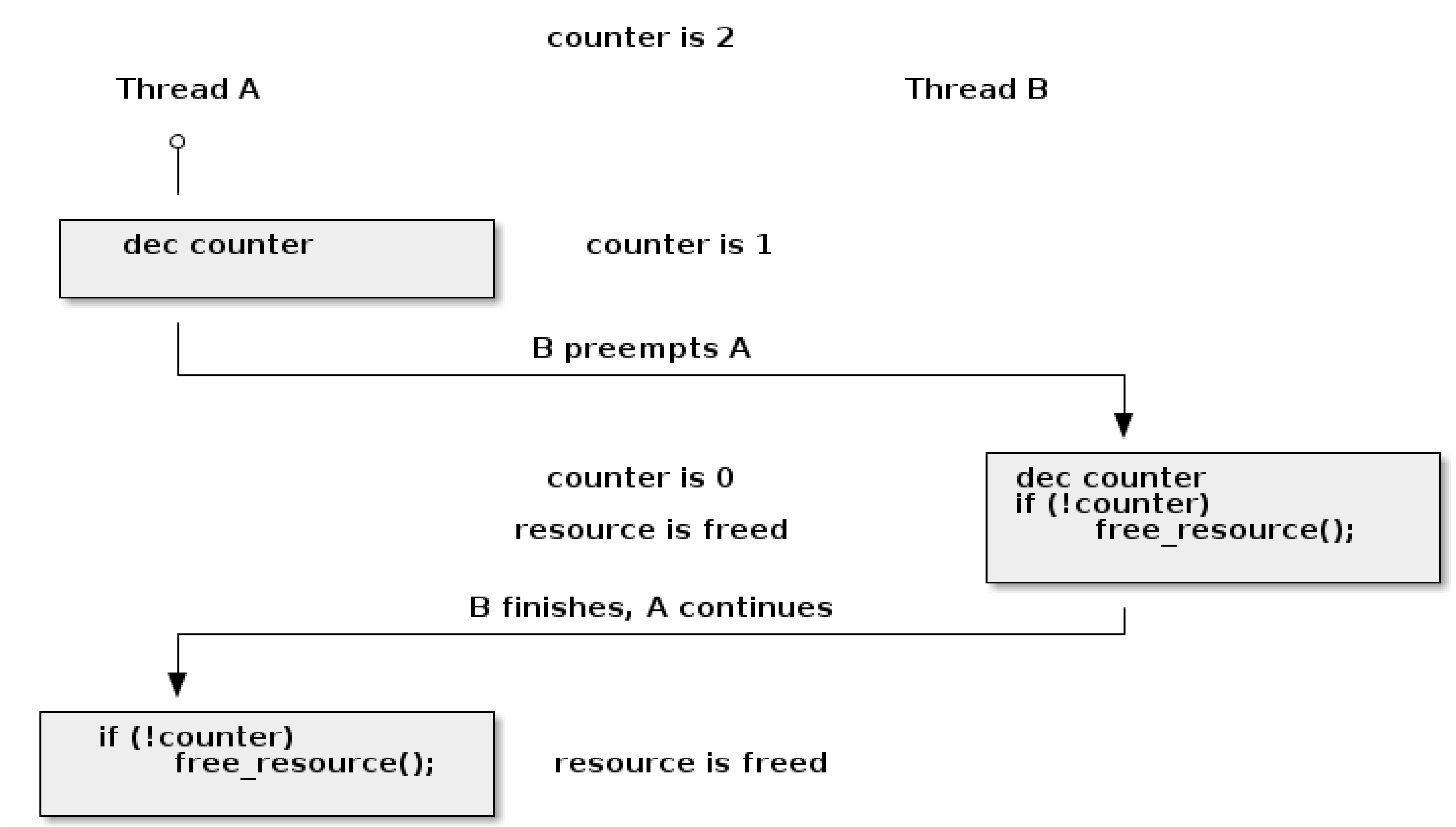
```
void down(struct semaphore * sem)
{ /* BEGIN CRITICAL SECTION */
--sem->count;
                  if (sem->count < 0) {   /* END CRITICAL SECTION */
                                   struct wait_queue wait = { current, NULL };
                                   current->state = TASK_UNINTERRUPTIBLE;
                                  add_wait_queue(&sem->wait, &wait);
                  for (;;) {
                                   unsigned long flags;
                                   spin_lock_irqsave(&semaphore_wake_lock, flags);
                   if (sem->waking > 0) {
                                   sem->waking--;
                                  break;
                   }
                  spin_unlock_irqrestore(&semaphore_wake_lock, flags);
                  schedule( );
                  current->state = TASK_UNINTERRUPTIBLE;
 }
                   spin_unlock_irqrestore(&semaphore_wake_lock, flags);
                  current->state = TASK_RUNNING;
                   remove_wait_queue(&sem->wait, &wait);  } }
```

- The function decrements the count field of the *sem semaphore, then checks whether its value is negative. The decrement and the test must be atomically executed, otherwise another kernel control path could concurrently access the field value, with disastrous results.
- Therefore, these two operations are implemented by means of the following assembly language instructions:

```
movl sem, %ecx
lock /* only for multiprocessor systems */
decl (%ecx)
js 2f
```

- On a multiprocessor system, the decl instruction is prefixed by a lock prefix to ensure the atomicity of the decrement operation.
- If count is greater than or equal to 0, the current process acquires the resource and the execution continues normally. Otherwise, count is negative and the current process must be, suspended. It is inserted into the wait queue list of the semaphore and put to sleep by directly invoking the schedule( ) function.

- The process is woken up when the resource is freed.
- Nonetheless, it cannot assume that the resource is now available, since several processes in the semaphore wait queue could be waiting for it.
- In order to select a winning process, the waking field is used: when the releasing process is going to wake up the processes in the wait queue, it increments waking; each awakened process then enters a critical region of the **down( )** function and tests whether waking is positive.
- If an awakened process finds the field to be positive, it decrements waking and acquires the resource; otherwise it goes back to sleep.
- The critical region is protected by the **semaphore_wake_lock global** spin lock and by interrupt disabling.



```
dec counter
```

```
dec counter
if (!counter)
        free_resource();
```

```
if (!counter)
        free_resource();
```

- A slightly different function called **down_interruptible( )** is also defined.
- It is widely used by device drivers since it allows processes that receive a signal while being blocked on a semaphore to give up the "down" operation.
- If the sleeping process is awakened by a signal before getting the needed resource, the function increments the count field of the semaphore and returns the value -EINTR.
- On the other hand, if **down_interruptible( )** runs to normal completion and gets the resource, it returns 0. The device driver may thus abort the I/O operation when the return value is -EINTR.
- When a process releases a kernel semaphore lock, it invokes the up( ) function.

```
         void up(struct semaphore * sem)
         {
                   /* BEGIN CRITICAL SECTION */
         ++sem->count;
         if (sem->count <= 0) {
                   /* END CRITICAL SECTION */
                   unsigned long flags;
                   spin_lock_irqsave(&semaphore_wake_lock, flags);
         if (atomic_read(&sem->count) <= 0)
                   sem->waking++;
                   spin_unlock_irqrestore(&semaphore_wake_lock, flags);
                   wake_up(&sem->wait);
         }
         }
}
```

Few examples of semaphores

- **Slab cache list semaphore -** The list of slab cache descriptors is protected by the **cache_chain_sem** semaphore, which grants an exclusive right to access and modify the list.
- A race condition is possible when **kmem_cache_create( )** adds a new element in the list, while **kmem_cache_shrink( )** and **kmem_cache_reap( )** sequentially scan the list.
- However, these functions are never invoked while handling an interrupt, and they can never block while accessing the list.
- Since the kernel is nonpreemptive, this semaphore plays an active role only in multiprocessor systems.
- **Memory descriptor semaphore -** Each memory descriptor of type mm_struct includes its own semaphore in the mmap_sem field.
- The semaphore protects the descriptor against race conditions that could arise because a memory descriptor can be shared among several lightweight processes.
- For instance, let us suppose that the kernel must create or extend a memory region for some process; it invokes the **do_mmap( )** function, which allocates a new **vm_area_struct** data structure.
- In doing so, the current process could be suspended if no free memory is available, and another process sharing the same memory descriptor could run.
- Without the semaphore, any operation of the second process that requires access to the memory descriptor could lead to severe data corruption.

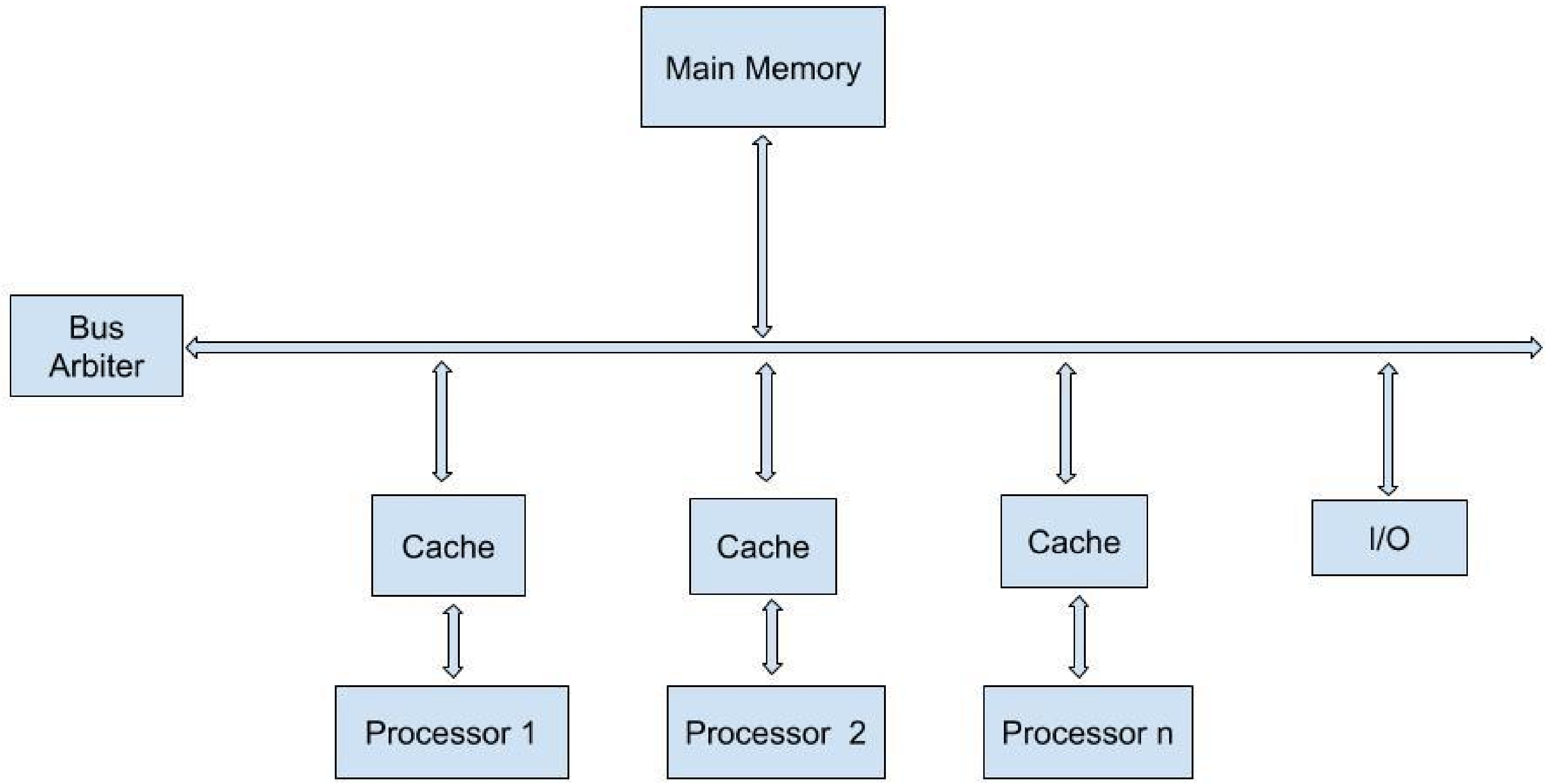
- **Inode semaphore** - Linux stores the information on a disk file in a memory object called an inode. The corresponding data structure includes its own semaphore in the **i_sem** field.
- A huge number of race conditions can occur during filesystem handling. Indeed, each file on disk is a resource held in common for all users, since all processes may access the file content, change its name or location, destroy or duplicate it, and so on.
- For example, let us suppose that a process is listing the files contained in some directory. Each disk operation is potentially blocking, and therefore even in uniprocessor systems other processes could access the same directory and modify its content while the first process is in the middle of the listing operation.
- Or again, two different processes could modify the same directory at the same time. All these race conditions are avoided by protecting the directory file with the inode semaphore.

Avoiding Deadlocks on Semaphores

- Whenever a program uses two or more semaphores, the potential for deadlock is present because two different paths could end up waiting for each other to release a semaphore.
- A typical deadlock condition occurs when a kernel control path gets the lock for semaphore A and is waiting for semaphore B, while another kernel control path holds the lock for semaphore B and is waiting for semaphore A.
- Linux has few problems with deadlocks on semaphore requests, since each kernel control path usually needs to acquire just one semaphore at a time.
- However, in a couple of cases the kernel must get two semaphore locks. This occurs in the service routines of the **rmdir( )** and the **rename( )** system calls.
- In order to avoid such deadlocks, semaphore requests are performed in the order given by addresses: the semaphore request whose semaphore data structure is located at the lowest address is issued first.

The SMP Architecture

- Symmetrical multiprocessing (SMP) denotes a multiprocessor architecture in which no CPU is selected as the Master CPU, but rather all of them cooperate on an equal basis, hence the name "symmetrical."
- From the kernel design point of view, an SMP kernel remains the same no matter how many CPUs are involved. The big jump in complexity occurs when moving from one CPU to two.
- Before proceeding in describing the changes that had to be made to Linux in order to make it a true SMP kernel, we shall briefly review the hardware features of the Pentium dualprocessing systems.
- These features lie in the following areas of computer architecture:
  - Shared memory
  - Hardware cache synchronization
  - Atomic operations
  - Distributed interrupt handling
  - Interrupt signals for CPU synchronization
- Some hardware issues are completely resolved within the hardware.



Main memory and data bus or I/O bus being shared among multiple processors in SMP

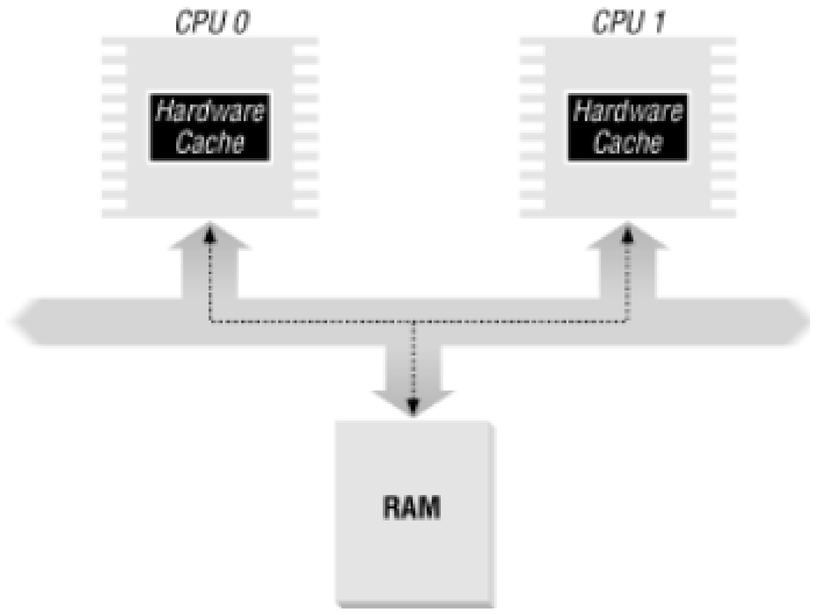
Common Memory

- All the CPUs share the same memory; that is, they are connected to a common bus.
- This means that RAM chips may be accessed concurrently by independent CPUs.
- Since read or write operations on a RAM chip must be performed serially, a hardware circuit called a memory arbiter is inserted between the bus and every RAM chip.
- Its role is to grant access to a CPU if the chip is free and to delay it if the chip is busy.
- Even uniprocessor systems make use of memory arbiters, since they include a specialized processor called DMA that operates concurrently with the CPU.

- In the case of multiprocessor systems, the structure of the arbiter is more complex since it has more input ports.
- The dual Pentium, for instance, maintains a two-port arbiter at each chip entrance and requires that the two CPUs exchange synchronization messages before attempting to use the bus.
- From the programming point of view, the arbiter is hidden since it is managed by hardware circuits.

Hardware Support to Cache Synchronization

- The section explains that the contents of the hardware cache and the RAM maintain their consistency at the hardware level.
- The same approach holds in the case of a dual processor. As shown in Figure below, each CPU has its own local hardware cache.
- But now updating becomes more time-consuming: whenever a CPU modifies its hardware cache it must check whether the same data is contained in the other hardware cache and, if so, notify the other CPU to update it with the proper value.
- This activity is often called cache snooping. Luckily, all this is done at the hardware level and is of no concern to the kernel.



SMP Atomic

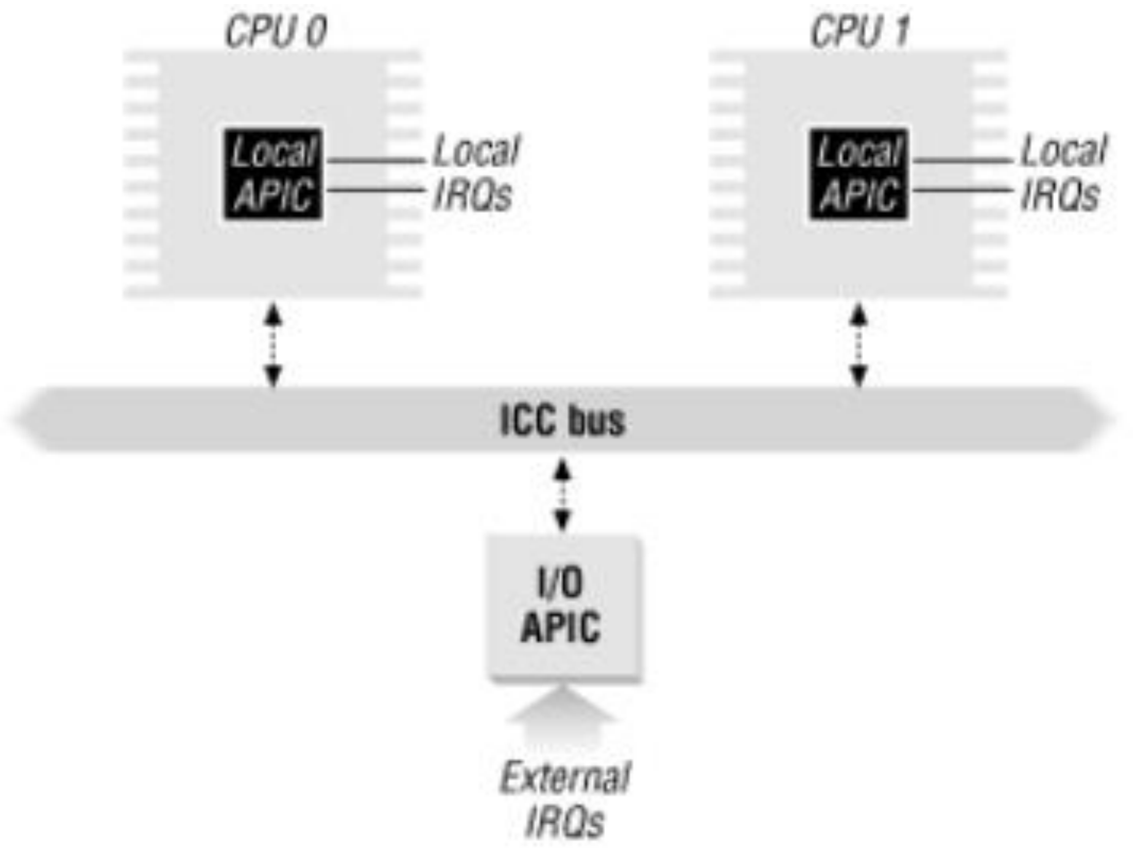
- Since standard read-modify-write instructions actually access the memory bus twice, they are not atomic on a multiprocessor system.
- Let us give a simple example of what might happen if an SMP kernel used standard instructions.
- Consider the semaphore implementation described earlier and assume that the **down( )** function decrements and tests the count field of the semaphore with a simple **decl** assembly language instruction.
- What happens if two processes running on two different CPUs simultaneously execute the decl instruction on the same semaphore?
- Well, decl is a read-modify-write instruction that accesses the same memory location twice: once to read the old value and again to write the new value.

- At first, both CPUs are trying to read the same memory location, but the memory arbiter steps in to grant access to one of them and delay the other.
- However, when the first read operation is complete the delayed CPU reads exactly the same (old) value from the memory location.
- Both CPUs then try to write the same (new) value on the memory location; again, the bus memory access is serialized by the memory arbiter, but eventually both write operations will succeed and the memory location will contain the old value decremented by 1.
- That gives, the global result is completely incorrect.
- For instance, if count was previously set to 1, both kernel control paths will simultaneously gain mutual exclusive access to the protected resource.
- Lock instruction prefixes have been introduced to solve that kind of problem. From the programmer's point of view, lock is just a special byte that is prefixed to an assembly language instruction.

- When the control unit detects a lock byte, it locks the memory bus so that no other processor can access the memory location specified by the destination operand of the following assembly language instruction.
- The bus lock is released only when the instruction has been executed
- Therefore, read-modify-write instructions prefixed by lock are atomic even in a multiprocessor environment.
- The Pentium allows a lock prefix on 18 different instructions. Moreover, some kind of instructions like xchg do not require the lock prefix because the bus lock is implicitly enforced by the CPU's control unit.

Distributed Interrupt Handling

- Being able to deliver interrupts to any CPU in the system is crucial for fully exploiting the parallelism of the SMP architecture.
- For that reason, Intel has introduced a new component designated as the I/O APIC (I/O Advanced Programmable Interrupt Controller), which replaces the old 8259A Programmable Interrupt Controller.



- Each CPU chip has its own integrated Local APIC.
- An Interrupt Controller Communication (ICC) bus connects a frontend I/O APIC to the Local APICs. The IRQ lines coming from the devices are connected to the I/O APIC, which therefore acts as a router with respect to the Local APICs.
- Each Local APIC has 32-bit registers, an internal clock, a timer device, 240 different interrupt vectors, and two additional IRQ lines reserved for local interrupts, which are typically used to reset the system.
- The I/O APIC consists of a set of IRQ lines, a 24-entry Interrupt Redirection Table, programmable registers, and a message unit for sending and receiving APIC messages over the ICC bus.
- Unlike IRQ pins of the 8259A, interrupt priority is not related to pin number: each entry in the Redirection Table can be individually programmed to indicate the interrupt vector and priority, the destination processor, and how the processor is selected. The information in the Redirection Table is used to translate any external IRQ signal into a message to one or more Local APIC units via the ICC bus.
- Interrupt requests can be distributed among the available CPUs in two ways:
    - **Fixed mode**
- The IRQ signal is delivered to the Local APICs listed in the corresponding Redirection Table entry.
    - **Lowest-priority mode**
- The IRQ signal is delivered to the Local APIC of the processor which is executing the process with the lowest priority. Any Local APIC has a programmable task priority.

Spin Locks

- Spin Locks are a locking mechanism are similar to the kernel semaphores described earlier, except that when a process finds the lock closed by another process, it "spins" around repeatedly, executing a tight instruction loop.
- Of course, spin locks would be useless in a uniprocessor environment, since the waiting process would keep running, and therefore the process that is holding the lock would not have any chance to release it.
- In a multiprocessing environment, however, spin locks are much more convenient, since their overhead is very small. In other words, a context switch takes a significant amount of time, so it is more efficient for each process to keep its own CPU and simply spin while waiting for a resource.
- Each spin lock is represented by a spinlock_t structure consisting of a single lock field; the values and 1 correspond, respectively, to the "unlocked" and the "locked" state. The SPIN_LOCK_UNLOCKED macro initializes a spin lock to 0.
- The functions that operate on spin locks are based on atomic read/modify/write operations; this ensures that the spin lock will be properly updated by a process running on a CPU even if other processes running on different CPUs attempt to modify the spin lock at the same time.

- The spin_lock macro is used to acquire a spin lock. It takes the address slp of the spin lock as its parameter and yields essentially the following code:

```
1: lock; btsl $0,
slp jnc 3f
2: testb $1,slp
 jne 2b
 jmp 1b 3:
```

- The btsl atomic instruction copies into the carry flag the value of bit in *slp, then sets the bit. A test is then performed on the carry flag: if it is null, it means that the spin lock was unlocked and hence normal execution continues at label 3 (the f suffix denotes the fact that the label is a "forward" one: it appear in a later line of the program). Otherwise, the tight loop at label 2 (the b suffix denotes a "backward" label) is executed until the spin lock assumes the value 0. Then execution restarts from label 1, since it would be unsafe to proceed without checking whether another processor has grabbed the lo

Table 11-3. Spin Lock Macros on a Multiprocessor System

Macro	Description
`spin_lock_init(slp)`	Set `slp->lock` to 0
`spin_trylock (slp)`	Set `slp->lock` to 1, return 1 if got the lock, otherwise
`spin_unlock_wait(slp)`	Cycle until `slp->lock` becomes 0
`spin_lock_irq(slp)`	`_ _cli( );spin_lock(slp)`
`spin_unlock_irq(slp)`	`spin_unlock(slp);__sti( )`
`spin_lock_irqsave(slp,flags)`	`__save_flags(flags);__cli( ); spin_lock(slp)`
`spin_unlock_irqrestore(slp,flags)`	`spin_unlock(slp);_ _restore_flags(flags)`