Kernal synchornization
- The kernel is like a server handling requests from processes and devices.
- These requests run concurrently and can create race conditions.
- This chapter covers synchronization techniques.
- It explores multiprocessor architecture and mutual exclusion in SMP Linux.

Kernel control path
- Kernel functions are executed in response to requests from User Mode processes or external devices via interrupts.
- These sequences of instructions in Kernel Mode are called "kernel control paths."
- Kernel control paths handle system calls, exceptions, or interrupts.
- Unlike processes, kernel control paths lack descriptors and are managed through instruction sequences.
- In some cases, the CPU runs a kernel control path sequentially, but it can interleave them during context switches or interrupts.
- Interleaving is crucial for multiprocessing and improves controller throughput.
- When interleaving, it's essential to protect critical data structures to prevent corruption.

Synchronization Techniques
- Race conditions and critical regions apply to kernel control paths as they do to processes.
- Race conditions arise when interleaved control paths affect the outcome of computations.
- Critical regions are sections of code that must be executed entirely by one control path before another can enter.
- Four synchronization techniques are discussed: nonpreemptability, atomic operations, interrupt disabling, and locking.

Nonpreemptability of Processes in Kernel Mode
- Linux's Kernel Mode is non-preemptive, meaning a running process in Kernel Mode won't be replaced by a higher-priority process unless it voluntarily gives up control.
- Interrupts or exceptions can interrupt a Kernel Mode process, but once the handler finishes, the original kernel control path resumes.
- Kernel control paths handling interrupts or exceptions can only be interrupted by other such control paths.
- Nonblocking system call control paths are atomic with respect to each other, simplifying kernel function implementations.
- When a process in Kernel Mode voluntarily yields the CPU, it must ensure data structures remain consistent.
- Upon resuming, the process must recheck previously accessed data structures that may have changed due to other control paths running the same code for different processes.

Atomic operations:
- Atomic operations are operations that execute in a single, uninterrupted assembly language instruction.
- They help prevent race conditions by ensuring that operations are completed without interruption at the chip level.
- Examples of atomic operations include instructions that make zero or one memory access.
- Read/modify/write instructions like "inc" or "dec" are atomic if no other processor accesses the memory between the read and write.
- Instructions prefixed with "lock" (0xf0) are atomic even on multiprocessor systems, as they lock the memory bus until the instruction is finished.

- Instructions prefixed with "rep" (0xf2, 0xf3) are not atomic because they check for interrupts between iterations.
- In the Linux kernel, special functions are used to ensure atomicity, with a lock byte added for multiprocessor systems.

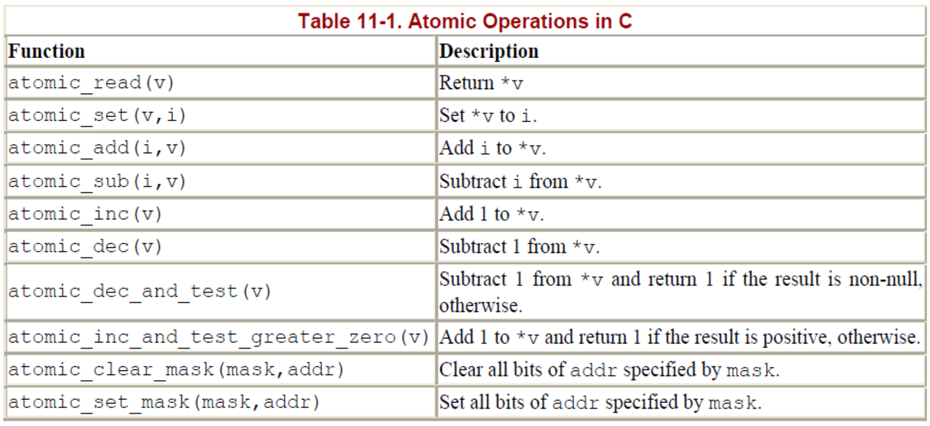


Table 11-1. Atomic Operations in C

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:

- Critical sections in code require more than atomic operations and typically include an atomic operation as their core.
- Interrupt disabling is a mechanism to create critical sections in the kernel, allowing a control path to run without interruption from hardware interrupts.
- However, interrupt disabling alone doesn't prevent all forms of control path interleaving.
- Examples of scenarios where interrupt disabling may not be sufficient include "Page fault" exceptions or when the schedule() function is invoked.
- In the Linux kernel, interrupt disabling is achieved using assembly instructions like cli and sti, or via macros like __cli and __sti.
- To save and restore the state of the IF flag in the eflags register, the kernel uses macros like __save_flags and __restore_flags.
- These macros help maintain proper interrupt handling, ensuring that interrupts are only enabled if they were enabled before the critical section.
- Linux provides additional synchronization macros, especially important on multiprocessor systems, but they are somewhat redundant on uniprocessor systems.

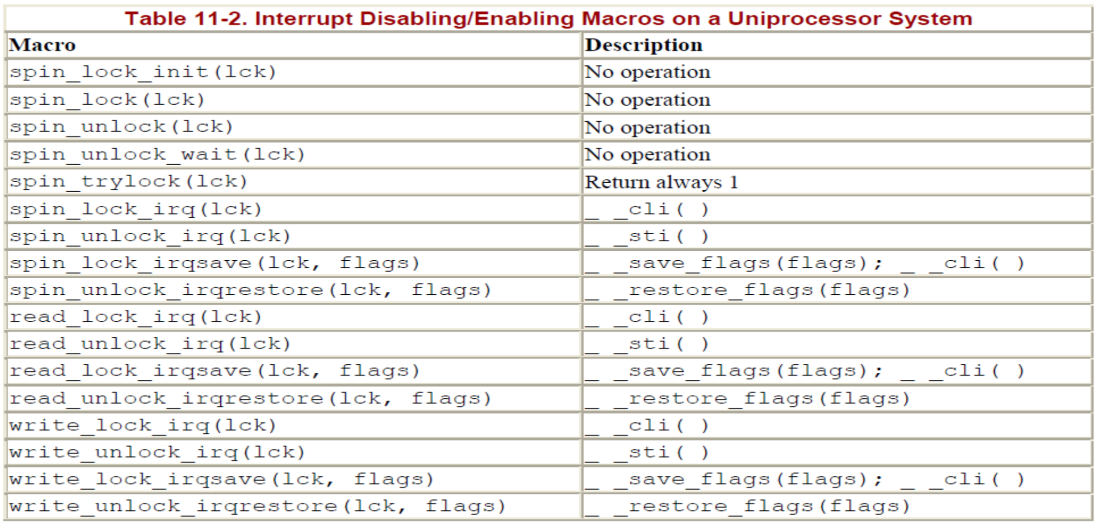


Table 11-2. Interrupt Disabling/Enabling Macros on a Uniprocessor System

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)

- `add_wait_queue()` and `remove_wait_queue()` functions protect the wait queue list with `write_lock_irqsave()` and `write_unlock_irqrestore()` functions.

- `setup_x86_irq()` adds a new interrupt handler for a specific IRQ and uses `spin_lock_irqsave()` and `spin_unlock_irqrestore()` to protect the handler list.

- `run_timer_list()` function safeguards dynamic timer data structures with `spin_lock_irq()` and `spin_unlock_irq()`.

- `handle_signal()` function protects the blocked field of the current process with `spin_lock_irq()` and `spin_unlock_irq()`.

Interrupt disabling is used in these functions for simplicity but is recommended for short critical regions, as it blocks communication between I/O devices and the CPU. Longer critical regions should be implemented using locking mechanisms.

locking through kernel semaphores:

- Locking is a common synchronization technique in the kernel. It's used when a control path needs to access a shared resource or enter a critical region.
- Kernel semaphores are one way to implement locking. They ensure that only one control path can access a resource at a time.
- Semaphores have fields like "count" to indicate resource availability and "wait" to store sleeping processes.
- When a process wants to acquire a semaphore-protected resource, it calls `down()`. This function decrements the count and suspends if the resource is busy.
- Waking up a process from suspension is controlled by the "waking" field, ensuring only one process succeeds in acquiring the resource.
- The `up()` function is used to release a semaphore lock. It increments the count, potentially waking up waiting processes.
- There are variations of `down()` like `down_trylock()` and `down_interruptible()` for specific use cases.
- Semaphores are mainly used to prevent race conditions during I/O disk operations and when interrupt handlers access kernel data structures.
- Spin locks are preferred in multiprocessor systems for more complex synchronization scenarios.

Different types of semaphores and avoiding deadlocks:

Semaphore Types:

1. **Slab Cache List Semaphore:** Protects the list of slab cache descriptors. It prevents race conditions when functions like `kmem_cache_create()` modify the list while others like `kmem_cache_shrink()` scan it. This semaphore is mainly relevant in multiprocessor systems.

2. **Memory Descriptor Semaphore:** Found in each `mm_struct` memory descriptor, this semaphore prevents race conditions that could occur when multiple lightweight processes share the same memory descriptor. For example, it ensures data consistency during memory region creation or extension using `do_mmap()`.

3. **Inode Semaphore:** In the context of filesystem handling, each inode data structure has an `i_sem` semaphore. It guards against race conditions when multiple processes access and modify files on disk simultaneously.

Avoiding Deadlocks:

- Deadlocks can occur when two or more semaphores are used because different paths may end up waiting for each other to release a semaphore.

- Linux typically avoids deadlock issues with semaphore requests because most kernel control paths acquire only one semaphore at a time.

- In cases where the kernel needs to acquire two semaphores, it follows a specific order: semaphores are requested in the order of their addresses, preventing potential deadlocks. This approach is used, for example, in `rmdir()` and `rename()` system calls when working with two inodes.

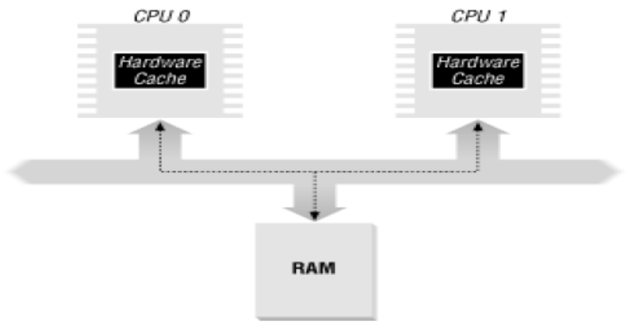
SMP architecture:

Symmetrical Multiprocessing (SMP):
- SMP is a multiprocessor architecture where all CPUs cooperate on an equal basis, without one being designated as a master CPU.
- The number of CPUs in an SMP system can vary, but it introduces challenges related to cache systems and cache synchronization.

Common Memory:
- SMP systems share the same memory, allowing multiple CPUs to access RAM concurrently.
- Memory arbiters control access to RAM chips, granting access to CPUs when the chips are available and delaying access when busy.

Hardware Cache Synchronization:
- Each CPU in an SMP system has its own local hardware cache.
- Cache synchronization ensures consistency between the cache and RAM. When a CPU modifies its cache, it checks if the same data is in another CPU's cache and notifies it to update with the correct value (cache snooping).
- These cache synchronization tasks are handled at the hardware level and are not a concern for the kernel.

Figure 11-1. The caches in a dual processor



Overall, an SMP kernel remains the same regardless of the number of CPUs in the system, simplifying kernel design and abstracting many hardware complexities.

In SMP (Symmetrical Multiprocessing) systems, several CPUs work together on an equal basis. Here are some key points related to SMP:

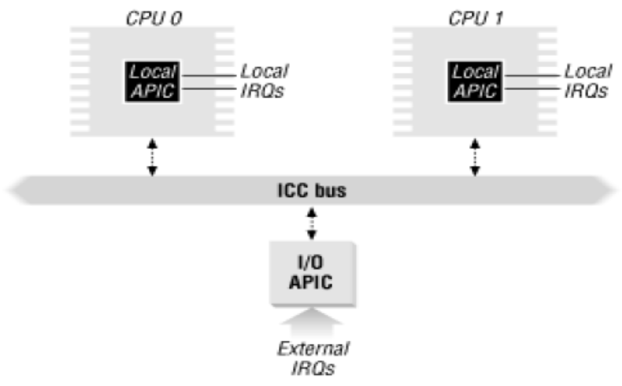
1. **Common Memory:** SMP systems share the same memory, and multiple CPUs can access RAM concurrently. Memory arbiters manage access to RAM chips to prevent conflicts.

2. **Hardware Cache Synchronization:** Each CPU has its own local hardware cache. Cache synchronization ensures that the cache contents remain consistent with RAM. This is done through cache snooping and is handled at the hardware level.

3. **SMP Atomic Operations:** Standard read-modify-write instructions are not atomic on multiprocessor systems. To ensure atomicity, the `lock` instruction prefix is used to lock the memory bus during such operations.

4. **Distributed Interrupt Handling:** SMP systems use the I/O APIC (I/O Advanced Programmable Interrupt Controller) to deliver interrupts to any CPU in the system. Each CPU has its own Local APIC, and an ICC (Interrupt Controller Communication) bus connects them to the I/O APIC. Interrupt requests can be distributed using fixed mode or lowest-priority mode. Interprocessor interrupts allow CPUs to send interrupts to each other.

Figure 11-2. APIC system



SMP systems introduce complexity due to cache synchronization and interrupt distribution, but the Linux kernel abstracts most of these hardware complexities, providing a unified kernel regardless of the number of CPUs in the system.

Kernel semaphores are represented as a struct with three fields:

1. `count`: An integer value that indicates whether a resource is available. If `count` is greater than 0, the resource is free and available. If `count` is less than or equal to 0, the semaphore is busy, and the absolute value of `count` represents the number of processes waiting for the resource. Zero means the resource is in use, but no other process is waiting for it.

2. `wait`: Stores the address of a wait queue list, containing sleeping processes waiting for the resource. If `count` is greater than or equal to 0, the wait queue is empty.

3. `waking`: Ensures that only one of the sleeping processes waiting for the resource succeeds in acquiring it when the resource becomes available.

These fields collectively help manage access to shared resources in the kernel, ensuring proper synchronization and control over resource availability.

Up()

```
function up(semaphore):
   semaphore.count++
   if semaphore.count <= 0:
      semaphore.waking++
      wake_up_one_process(semaphore.wait)
```