Roll no: 2020201026

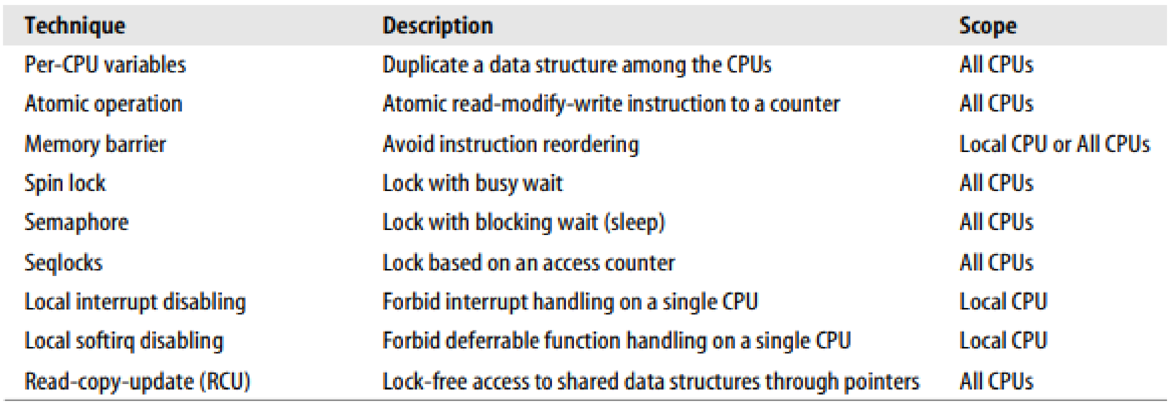
Topic: Kernel Synchronization

Need of Synchronization [i][ii]

In a multiprogramming and multiprocessing system, sharing of common data between multiple processes and implementing a re-entrant kernel requires the use of synchronization. Without proper synchronization, the interaction of multiple processes or kernel control paths may lead to the corruption of stored information. When two or more kernel control paths use the same resource item and the outcome of a computation depends on how two or more processes are scheduled, we say that there is a race condition. Any section of code that should be finished by each process that begins it before another process can enter it, is called a critical region.

Kernel Synchronization [iii][iv][v][vi]

The following information describes several synchronization techniques used by the Linux kernel to interleave its control paths by avoiding race conditions among shared data.



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 (RCU)	Lock-free access to shared data structures through pointers	All CPUs

1. Per-CPU Variables

A per-CPU variable is an array of kernel data structures, of which one element is allotted per CPU. It is the simplest synchronization technique that declares the kernel variables as per-CPU variables, where a CPU cannot access the elements of the array corresponding to other CPUs. It can freely read and modify its own element without the fear of race conditions as it is the only CPU having the right to do so. This technique can be used only in particular cases where it makes sense to split data across multiple CPUs of the system. Although per-CPU variables provide protection against concurrent accesses from several CPUs, they do not provide protection against access from asynchronous functions such as interrupt handlers. Also, per-CPU variables are prone to race conditions caused by kernel preemption, both in uniprocessor and multiprocessor systems. Thus, additional synchronization primitives are required to handle these cases.

2. Atomic Operations

Several assembly language instructions are of type read-modify-update where memory location is accessed twice – first for reading and then for writing the new value. If two kernel control paths try to read-modify-write the same memory location at the same time by executing non-atomic operations, their interleaving my lead to incorrect result as one control path would over-write the other's result. To avoid this problem, such operations should be kept atomic at the chip level. Every read-modify-write operation should be executed as a single instruction without being interrupted in the middle. Few such instructions in assembly language are:

- Assembly language instructions that make zero or one aligned memory access are atomic.
- INC and DEC instructions perform memory read, update and write operations atomically if now other processor steals the memory bus between read and write operations.
- Instructions whose opcode is prefixed by the lock byte (0xf0) are atomic even on a multiprocessor system as it locks the memory bus to prevent memory bus stealing until the instruction execution is completed.

When writing a C code, we cannot guarantee that the compiler will use an atomic instruction for the operations. 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.

3. Optimization and Memory Barriers

Compilers usually reorder the assembly language instructions to optimize how registers are used and also modern CPUs usually execute several instructions in parallel by reordering memory accesses. This reordering of instructions should be avoided during synchronization as it might happen that 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 barrier: It ensures that the instructions placed before the primitive are not mixed by the compiler with the instructions placed after the primitive. In Linux the barrier() macro, which expands into asm volatile(""::: "memory") acts as an optimization barrier. 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.

Memory barrier: It ensures that that the operations placed before the primitive are finished before starting the operations placed after the primitive. Few assembly language instructions which act as memory barriers in 80x86 processors are:

- 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
- The lfence, sfence, and mfence assembly language instructions that implement read memory barriers, write memory barriers, and read-write memory barriers, respectively.
- A few special assembly language instructions such as iret that terminates an interrupt or exception handler

Linux provides both read-memory barriers that act only on instructions that read from memory, as well as write-memory barriers that act only on instructions that write to memory. xmb() primitives are used whenever race conditions are to be prevented on both uniprocessor and multiprocessor systems, while smp_xxx() primitives are used only in multiprocessor systems.

4. Spin Locks

When a kernel control path needs to access a shared data structure or enter a critical region, it must acquire a lock for it. 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 another kernel control path, it spins around repeatedly in a loop until the lock is released. The waiting kernel control path keeps running on the CPU in a busy wait state. 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. Spin locks, ironically, are global and therefore must themselves be protected against concurrent accesses. In Linux, each spin lock is represented by a spinlock_t structure consisting of two fields: slock (encodes the spin lock state) and break_lock (signals if a process is busy waiting for the lock).

Read/write spin locks: They increase the amount of concurrency inside the kernel by allowing 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. Each read/write spin lock is a rwlock_t structure which consists of a counter denoting the number of control paths currently reading the protected data structure and an unlock flag that is set when no kernel control path is reading or writing.

5. Seqlocks

The read/write spin lock requests issued by the kernel control paths gave same priority to read_lock or write_lock operations. Seqlocks which are introduced in Linux 2.6 are similar to read/write spin locks, but they give higher priority to the writers. A writer never has to wait as he is allowed to proceed even when readers are active. Each seqlock is a seqlock_t structure consisting of two fields: a lock field of type spinlock_t and an integer sequence field. Whenever a new writer becomes active, it increments the sequence counter. Each reader reads this sequence counter twice before and after reading the data. If both times the count is the same then the data is valid, but in other case, it means that a new writer has entered and thus the data just read is not valid anymore. A data structure can be protected by a seqlock only if it does not include pointers that are modified by the writers and dereferenced by the readers and the code in the critical regions of the readers does not have side effects. Furthermore, the critical regions of the readers should be short and writers should seldom acquire the seqlock, otherwise repeated read accesses would cause a severe overhead.

6. Read-Copy Update (RCU)

RCU is a new addition in Linux 2.6 that is mostly used in the networking layer and in the Virtual Filesystem. Unlike seqlocks, RCU allows many readers and many writers to proceed concurrently. To avoid the overhead caused by read/write spin locks and seqlocks due to cache line-snooping and invalidation, RCU uses no lock or counter shared by all CPUs, that is it is lock-free. The scope of RCU is limited to data structures that are dynamically allocated and referenced by means of pointers, and, kernel control path that does not sleep in its critical section. In order to read an RCU-protected data structure, the kernel control path executes the rcu_read_lock() macro which is equivalent to preempt_disable(). Then the reader dereferences the pointer to the data structure and starts reading it. The end of the critical region is marked by the rcu_read_unlock() macro, which is equivalent to preempt_enable().

When a writer wants to update the data structure, it dereferences the pointer and makes a copy of the whole data structure and modifies the copy. After modifying, the writer changes the pointer to the data structure so as to make it point to the updated copy. As this is an atomic operation, no corruption in the data structure may occur. But to ensure that the updated pointer is seen by the other processes only after the data structure has been modified, a memory barrier is required. The problem with the RCU technique is that the old copy of the data structure cannot be freed right away when the writer updates the pointer as the readers that were accessing the data structure when the writer started its update could still be reading the old copy. The old copy can be freed only after all (potential) readers on the CPUs have executed the rcu_read_unlock() macro. The call_rcu() function is invoked by the writer to get rid of the old copy of the data structure.

7. Kernel Semaphores

Semaphores are sleeping locks. 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, which means interrupt handlers and deferrable functions cannot use them. The up() and down() functions are used operate on these semaphores to request or release the protected resource. A kernel semaphore is an object of type struct semaphore, containing the fields:

- Count: It stores and atomic_t value which if > 0 means that the resource is free and available. If count is = 0, means that the resource busy and no other process is waiting on it. Else if count < 0, means that the resource is unavailable and at least one process 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.
- Sleepers: Stores a flag that indicates whether some processes are sleeping on the semaphore

Read/write semaphores: They are similar to the 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. When the semaphore is released, the process in the first position of the wait queue list is checked and 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 read lock. Each read/write semaphore is described by a rw_semaphore structure that includes:

- Count: Stores two 16-bit counters. The counter in the MSW encodes the sum of the number of nonwaiting writers and the number of waiting kernel control paths. The counter in the LSW encodes the total number of nonwaiting readers and writers.
- Wait_list: Points to a list of waiting processes which has pointers to the descriptors of the sleeping processes and a flag indicating whether the process wants the semaphore for reading or for writing.
- Wait_lock: A spin lock used to protect the wait queue list and the rw_semaphore structure itself.

8. Completions

Linux 2.6 makes use of another synchronization primitive called completions, which are similar to semaphores. According to semaphore implementation, up() and down() can execute concurrently on the same semaphore. The issue with this is for example that if one process destroys the semaphore and the other process executes an up() on the same semaphore, it might attempt to access a data structure that no longer exists. Completions are specifically designed to solve this problem without the need to add more instructions in the up() and down() functions as that would be a bad thing to do for functions that are so heavily used. The function corresponding to up() is called complete() and the function corresponding to down() is called wait_for_completion(). The real difference between completions and semaphores is how the spin lock included in the wait queue is used. In completions, the spin lock is used to ensure that complete() and wait_for_completion() cannot execute concurrently. In semaphores, the spin lock is used to avoid letting concurrent down()'s functions mess up the semaphore data structure.

9. 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. When the kernel enters a critical section, it disables interrupts by clearing the IF flag of the eflags register. Kernel saves the old setting of the flag by and at the end of the critical region, the saved contents of eflags is restored.

[i] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[ii] https://www.ibm.com/developerworks/library/l-linux-synchronization/index.html
[iii] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[iv] https://kerneltweaks.wordpress.com/2015/03/20/quick-guide-for-choosing-correct-synchronization-mechanism-inside-linux-kernel/
[v] https://notes.eddyerburgh.me/operating-systems/linux/kernel-synchronization
[vi] https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch05.html