CS343 - Operating Systems

QUIZ-3

Process Synchronization In Linux

Kartikeya Saxena
Roll No.: 180101034
kartikey18a@iitg.ac.in
Indian Institute Of Technology, Guwahati, Assam.

INTRODUCTION

- Linux uses various synchronization primitives to prevent race conditions on accessing shared resources
- Types of synchronization techniques:
  - Per CPU Variables
  - Atomic Operations
  - Optimization Barriers
  - Spin Locks
  - Seq Locks
  - Semaphores
- Each technique has its unique pros and cons.

PER CPU VARIABLES

- The best synchronization technique consists in designing so as to avoid the need for synchronization in the first place.
- A per-CPU variable is an array of data structures, one element per each CPU in the system.
- A CPU shouldn’t access the array element corresponding to other CPUs.
- This ensures that a CPU can change its own array element without any race condition.
- This technique has a very limited use and it is not synchronization in the true sense.

ATOMIC OPERATIONS

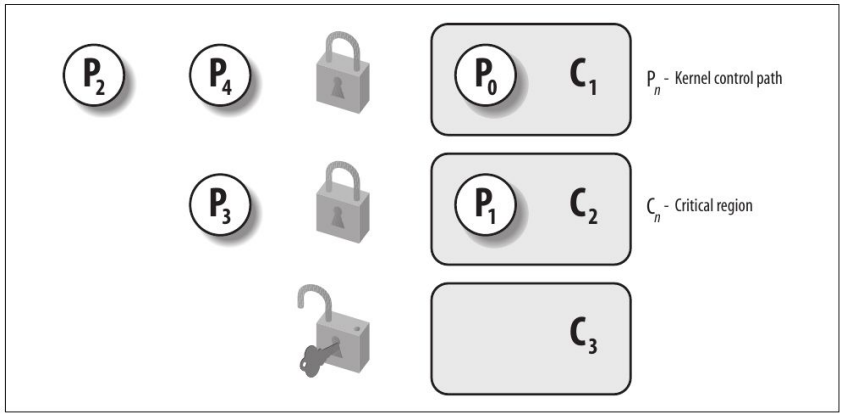
- An atomic instruction guarantees that the entire operation is not interleaved with any other CPU.
- Since architectures differ greatly, you'll find varying implementations of the atomic functions. Some are provided almost entirely in assembly, while others resort to C and disabling interrupts using local_irq_save and local_irq_restore.
- In x86 certain instructions can have a ‘lock’ prefix which ‘locks’ all of memory for this CPU.
- This is the easiest way to prevent race conditions due to “read-modify-write” instructions such as (inc or dec).

OPTIMIZATION BARRIER

- When using optimizing compilers, the compiler might reorder the assembly language instructions in such a way to optimize how registers are used.
- When dealing with synchronisation, this reorganization must not be taken for granted since it could lead to a ton load of problems!
- An optimization barrier primitive ensures that such intermingling of assembly language don't take place.
- In Linux it is achieved by barrier() macro which expands to asm volatile(“”:::”memory”)
  - volatile: forbids compiler to shuffle the asm instruction with other instructions of program
  - memory: instructs the compiler to assume that all memory locations in RAM are changed by this instruction

SPIN LOCKS

- Spin locks are a special kind of lock to protect critical sections which are 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 otherwise if it finds the lock “closed”, it “spins” around, repeatedly executing a loop (called “busy wait”).

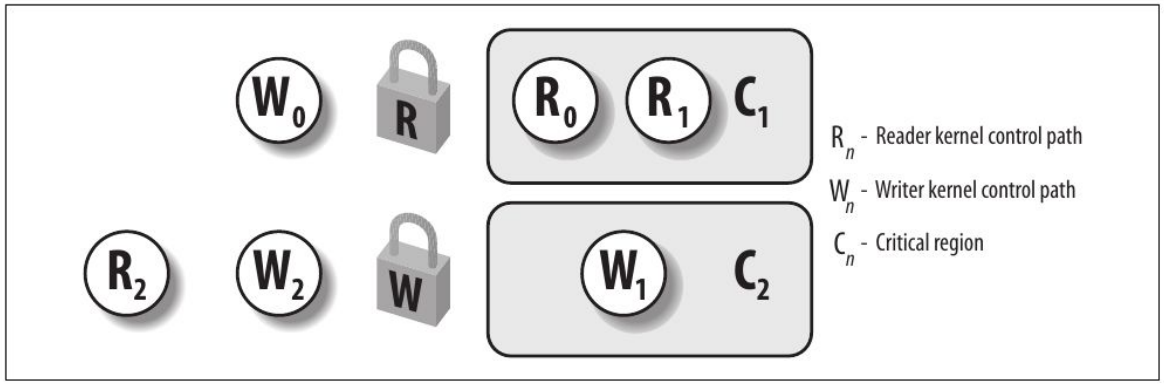


SPIN LOCKS

- Kernel preemption is disabled in critical paths protected by spinlocks
- The code that has taken the lock must not sleep as it wastes the current processor's time.
- Spin locks wastes time but are usually convenient as resources are locked for a fraction of a millisecond only.
- In Linux, each spinlock is represented by a spinlock_t structure:
  - slock : spin lock state (1 = unlocked, 0 = locked)
  - break_lock : flag indicating busy waiting for the lock
- **Spin lock macros**
  - spin_lock_init()
  - spin_lock()
  - spin_unlock()
  - spin_unlock_wait()
  - spin_trylock()

READ/WRITE SPIN LOCKS

- Read/write spin locks are introduced to increase system performance
- It allows several control paths to simultaneously read the shared data as long as no control path modifies it.
- If a control path wishes to modify the shared data it must acquire exclusive access to the resource.



READ/WRITE SPIN LOCKS

- In Linux, each read/write spinlock is represented by a rwlock_t structure:
  - 24-bit counter : denoting number of control paths reading the shared data. It is generally stored in the two's complement format
  - unlock flag: 24th bit indicating whether any control path is reading/writing the shared data

0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	f	f	f	f	f	f
0	0	f	f	f	f	f	e

- Idle lock
- Writer (flag clear and no Reader)
- 1 Reader
- 2 Readers

- **Read/write spinlock macros**
  - rwlock_init()
  - read_lock() / read_unlock()
  - rw_lock() / rw_unlock()

SEQ LOCKS

- In read/write spin locks both readers and writers have same priority and the writer must wait until all readers have finished. This could starve the writer.
- So seq locks were introduced in Linux to give higher priority to writer.
- In Linux, each seqlock is represented by a seqlock_t structure:
  - lock: type spinlock_t
  - sequence: Each reader must read this sequence twice, before and after reading the data, and check whether the two values coincide.
- The critical regions of the readers should be short and writers should seldom acquire the seqlocks, otherwise, readers could starve.

SEMAPHORES

- A Semaphore is similar to a spin lock, except for the fact that whenever a process tries to acquire a locked source instead of busy waiting, it is suspended. It becomes runnable again when the resource is released.
- In Linux, each semaphore is represented by a struct semaphore
  - count: if greater than 0 resource is free, busy otherwise
  - wait: stores address of all sleeping process waiting for the resource
  - sleepers: indicates whether any process are sleeping on the semaphore or not
- Linux uses functions up() and down()
  for releasing and getting resources
  and init_MUTEX() and init_MUTEX_LOCKED()
  for initializing semaphores.

SEMAPHORES

up()

```
movl $sem->count,%ecx
lock; incl (%ecx)
jg 1f
lea %ecx,%eax
pushl %edx
pushl %ecx
call __up
popl %ecx
popl %edx
1:
```

down()

```
movl $sem->count,%ecx
lock; decl (%ecx);
jns 1f
lea %ecx, %eax
pushl %edx
pushl %ecx
call __down
popl %ecx
popl %edx
1:
```

SEMAPHORES

__up()

```
__attribute__((regparm(3))) void
__up(struct semaphore *sem)
{
wake_up(&sem->wait);
}
```

__down()

```
__attribute__((regparm(3))) void __down(struct semaphore * sem)
{
        DECLARE_WAITQUEUE(wait, current);
        unsigned long flags;
        current->state = TASK_UNINTERRUPTIBLE;
        spin_lock_irqsave(&sem->wait.lock, flags);
        add_wait_queue_exclusive_locked(&sem->wait, &wait);
        sem->sleepers++;
        for (;;) {
                if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
                        sem->sleepers = 0;
                        Break; }
                sem->sleepers = 1;
                spin_unlock_irqrestore(&sem->wait.lock, flags);
                schedule();
                spin_lock_irqsave(&sem->wait.lock, flags);
                current->state = TASK_UNINTERRUPTIBLE;}
        remove_wait_queue_locked(&sem->wait, &wait);
        wake_up_locked(&sem->wait);
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        current->state = TASK_RUNNING;
}
```

WHICH ONE TO USE WHEN?

- Each technique has its various pros and cons

TECHNIQUE	PROS AND CONS
PER CPU VARIABLES	Simple, Avoids synchronization, limited use
ATOMIC INSTRUCTIONS	Makes inc and dec atomic, locks memory locations
OPTIMIZATION BARRIERS	Avoids reordering, prevents instruction optimization
SPIN LOCKS	Simple, wastes cycles in busy waiting
SEQ LOCKS	Reader starvation, improves writers performance
SEMAPHORES	Extra overhead in waiting queue, avoids busy waiting

Kartikeya Saxena

kartikey18a@iitg.ac.in