

Synchronization Mechanisms and Policies

Linux Device Drivers Chapter 5

Linux Kernel Development Chapters 9 and 10

Concurrency: The reason why synchronization is needed

Race conditions occur due to shared access to resources

- **[Interrupt]** A thread/process was working on some data when interrupt happened. The interrupt handler needs to work on the same data.
- **[Preemption]** A thread/process was working on some data when it was preempted by the kernel scheduler. The new thread/process needs to work on the same data.
- **[SMP or Symmetric Multiprocessing]** While a thread/process is working with some data, another thread/process needs to work on the same data on a different processor.

Critical Sections in Code

Synchronization: Only one thread of execution should enter the critical section at a time, other threads should wait.

- Taste food
- Add salt

Thread 1: Taste food
Thread 2: Taste food
Thread 1: Add salt
Thread 2: Add salt

Thread 2: Taste food
Thread 1: Taste food
Thread 1: Add salt
Thread 2: Add salt

Thread 2: Taste food
Thread 1: Taste food
Thread 2: Add salt
Thread 1: Add salt

Thread 1: Taste food
Thread 1: Add salt
Thread 2: Taste food
Thread 2: Add salt

Thread 1: Taste food
Thread 2: Taste food
Thread 2: Add salt
Thread 1: Add salt

Thread 2: Taste food
Thread 2: Add salt
Thread 1: Taste food
Thread 1: Add salt

How to make a thread wait (mechanisms)?

- Semaphores (counting or binary)
- Spinlocks
- Read/write semaphores/spinlocks
- Sequential lock
- Completion variables
- Lock free mechanisms: no waiting
- Read copy update (RCU)
- !!!!

Not all critical sections are the same

- Do we just add two integers or bits in the critical section?

It makes sense to use atomic instructions

- Do we sleep in the critical section or is the critical section too long?

- kcalloc()
- copy data to/from user space to kernel space

It makes sense to put the waiting thread(s) on sleep

- Is the critical section too small?

- Will context switching overhead be higher than the time it takes for the waiting thread to complete the critical section?

It makes sense to keep the waiting thread hold the cpu

- Do we have both readers and writers to the shared data?

It makes sense to synchronize only the writers, and the readers can share the resource simultaneously.

Atomic Integer Operation Instructions

(same present for 32 bits)

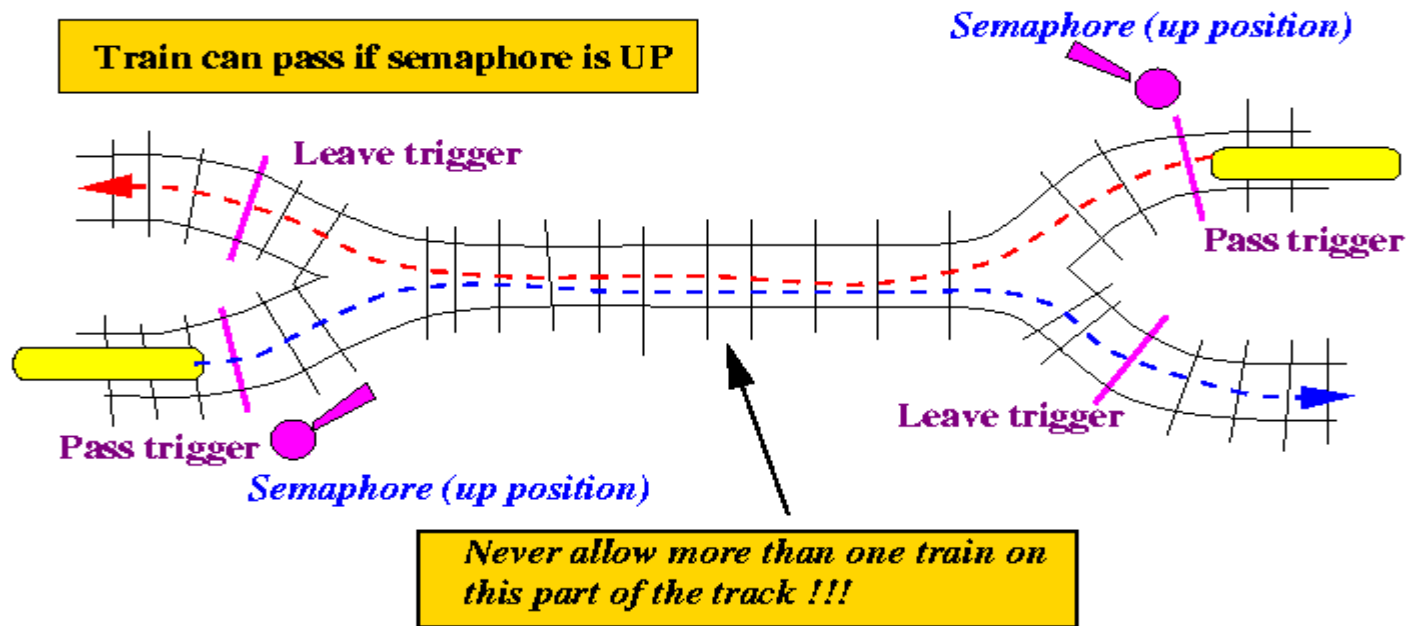
Atomic Integer Operation	Description
<code>ATOMIC64_INIT(long i)</code>	At declaration, initialize to <code>i</code> .
<code>long atomic64_read(atomic64_t *v)</code>	Atomically read the integer value of <code>v</code> .
<code>void atomic64_set(atomic64_t *v, int i)</code>	Atomically set <code>v</code> equal to <code>i</code> .
<code>void atomic64_add(int i, atomic64_t *v)</code>	Atomically add <code>i</code> to <code>v</code> .
<code>void atomic64_sub(int i, atomic64_t *v)</code>	Atomically subtract <code>i</code> from <code>v</code> .
<code>void atomic64_inc(atomic64_t *v)</code>	Atomically add one to <code>v</code> .
<code>void atomic64_dec(atomic64_t *v)</code>	Atomically subtract one from <code>v</code> .
<code>int atomic64_sub_and_test(int i, atomic64_t *v)</code>	Atomically subtract <code>i</code> from <code>v</code> and return true if the result is zero; otherwise false.
<code>int atomic64_add_negative(int i, atomic64_t *v)</code>	Atomically add <code>i</code> to <code>v</code> and return true if the result is negative; otherwise false.
<code>long atomic64_add_return(int i, atomic64_t *v)</code>	Atomically add <code>i</code> to <code>v</code> and return the result.
<code>long atomic64_sub_return(int i, atomic64_t *v)</code>	Atomically subtract <code>i</code> from <code>v</code> and return the result.
<code>long atomic64_inc_return(int i, atomic64_t *v)</code>	Atomically increment <code>v</code> by one and return the result.
<code>long atomic64_dec_return(int i, atomic64_t *v)</code>	Atomically decrement <code>v</code> by one and return the result.
<code>int atomic64_dec_and_test(atomic64_t *v)</code>	Atomically decrement <code>v</code> by one and return true if zero; false otherwise.
<code>int atomic64_inc_and_test(atomic64_t *v)</code>	Atomically increment <code>v</code> by one and return true if the result is zero; false otherwise.

Atomic Bit Operation Instructions

Atomic Bitwise Operation	Description
<code>void set_bit(int nr, void *addr)</code>	Atomically set the <i>nr-th</i> bit starting from <code>addr</code> .
<code>void clear_bit(int nr, void *addr)</code>	Atomically clear the <i>nr-th</i> bit starting from <code>addr</code> .
<code>void change_bit(int nr, void *addr)</code>	Atomically flip the value of the <i>nr-th</i> bit starting from <code>addr</code> .
<code>int test_and_set_bit(int nr, void *addr)</code>	Atomically set the <i>nr-th</i> bit starting from <code>addr</code> and return the previous value.
<code>int test_and_clear_bit(int nr, void *addr)</code>	Atomically clear the <i>nr-th</i> bit starting from <code>addr</code> and return the previous value.
<code>int test_and_change_bit(int nr, void *addr)</code>	Atomically flip the <i>nr-th</i> bit starting from <code>addr</code> and return the previous value.
<code>int test_bit(int nr, void *addr)</code>	Atomically return the value of the <i>nr-th</i> bit starting from <code>addr</code> .

Semaphore

(Dijkstra designed this inspired by trains)



Wait:

```
wait(S) {  
    while S <= 0  
        ; // no-op  
    S--;  
}
```

Signal:

```
signal(S) {  
    S++;  
}
```

Method

`down_interruptible (struct semaphore *)`

`down(struct semaphore *)`

`down_trylock(struct semaphore *)`

`up(struct semaphore *)`

Description

Tries to acquire the given semaphore and enter interruptible sleep if it is contended

Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended

Tries to acquire the given semaphore and immediately return nonzero if it is contended

Releases the given semaphore and wakes a waiting task, if any

Counting and Binary Semaphore

- Semaphores allow for an **arbitrary number of simultaneous lock holders**.
- The number of permissible simultaneous holders of semaphores can be set at declaration time. This value is called the **usage count** or simply the count.
- If the count is equal to one, the semaphore is called a **binary semaphore** (because it is either held by one task or not held at all) or a **mutex** (because it enforces mutual exclusion).
- If the count is initialized to a nonzero value greater than one, the semaphore is called a **counting semaphore**, and it enables at most count holders of the lock at a time.
- Counting semaphores are not used to enforce mutual exclusion because they enable multiple threads of execution in the critical region at once. Instead, they are used to **enforce limits in certain code**. They are not used much in the kernel.


Spinlocks

```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

Spinlocks

```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

Two threads reach here simultaneously?



Spinlocks

```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

What things are allowed for a thread waiting for a lock?
- can it be preempted by another process or interrupt?

Two threads reach here simultaneously?



Spinlocks

```
int lock = 0;
void spinlock(void)
{
    while (lock == 1)
        {};
    lock = 1;
}
```

What things are allowed for a thread holding a lock?
- can it be preempted by another process or interrupt?

What things are allowed for a thread waiting for a lock?
- can it be preempted by another process or interrupt?

Two threads reach here simultaneously?

Method	Description
<code>spin_lock()</code>	Acquires given lock
<code>spin_lock_irq()</code>	Disables local interrupts and acquires given lock
<code>spin_lock_irqsave()</code>	Saves current state of local interrupts, disables local interrupts, and acquires given lock
<code>spin_unlock()</code>	Releases given lock
<code>spin_unlock_irq()</code>	Releases given lock and enables local interrupts
<code>spin_unlock_irqrestore()</code>	Releases given lock and restores local interrupts to given previous state
<code>spin_lock_init()</code>	Dynamically initializes given <code>spinlock_t</code>
<code>spin_trylock()</code>	Tries to acquire given lock; if unavailable, returns nonzero
<code>spin_is_locked()</code>	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Semaphore vs. Spinlock: Policy

- Because the contending tasks sleep while waiting for the lock to become available, **semaphores are well suited to locks that are held for a long time.**
- **Semaphores are not optimal for locks that are held for short periods** because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.
- Unlike spin locks, semaphores do not disable kernel preemption and, consequently, **code holding a semaphore can be preempted.**

Reading vs. Writing Locks

- Threads should be able to read shared data simultaneously
- A writer thread needs to synchronize with readers and other writers
- Define separate locks for readers and writers

Sequential Lock

- Data has lot of readers and few writers
- Though writers are few, we favor them. Readers should never starve writers.
- No lock for readers, but a sequential variable read to match values..
- Lock for writers, immediately available

```
seqlock_t mr_seq_lock = DEFINE_SEQLOCK(mr_seq_lock);
```

```
write_seqlock(&mr_seq_lock);  
/* write lock is obtained... */  
write_sequnlock(&mr_seq_lock);
```

```
unsigned long seq;  
  
do {  
    seq = read_seqbegin(&mr_seq_lock);  
    /* read data here ... */  
} while (read_seqretry(&mr_seq_lock, seq));
```


Prominent Example of Sequential Lock: Linux Jiffies

- Jiffies hold a 64-bit count of the number of clock ticks since the machine booted.
- Only one writer (the timer interrupt)
- Enormous number of readers

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xtime_lock, seq));
    return ret;
}
```

```
write_seqlock(&xtime_lock);
jiffies_64 += 1;
write_sequnlock(&xtime_lock);
```

Read Copy Update (RCU)

- Reads are common, writes are rare
- Reads are lock free
- Resources being protected should be accessed via pointers, and all references to those resources must be held only by atomic code.
- When the data structure needs to be changed, the writing thread makes a copy, changes the copy, then aims the relevant pointer at the new version—thus, the name of the algorithm.
- When the kernel is sure that no references to the old version remain, it can be freed.

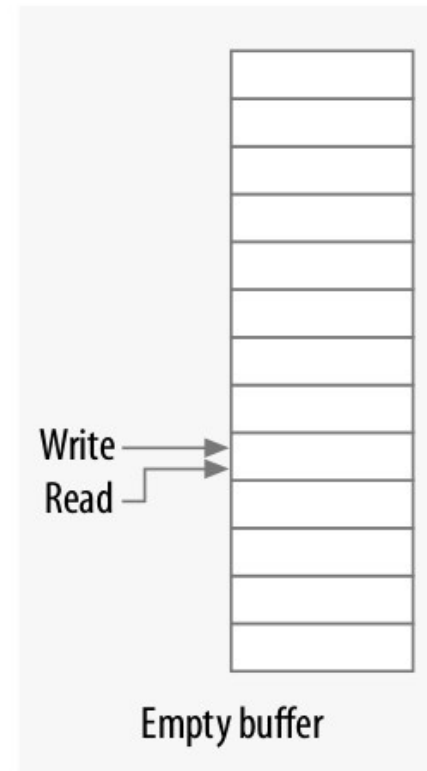
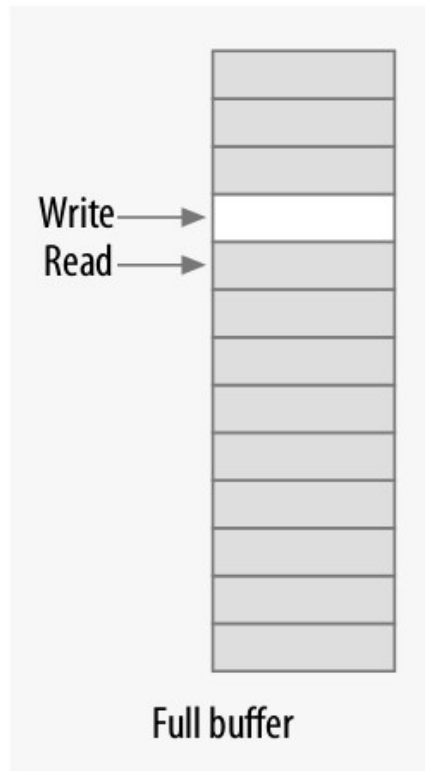
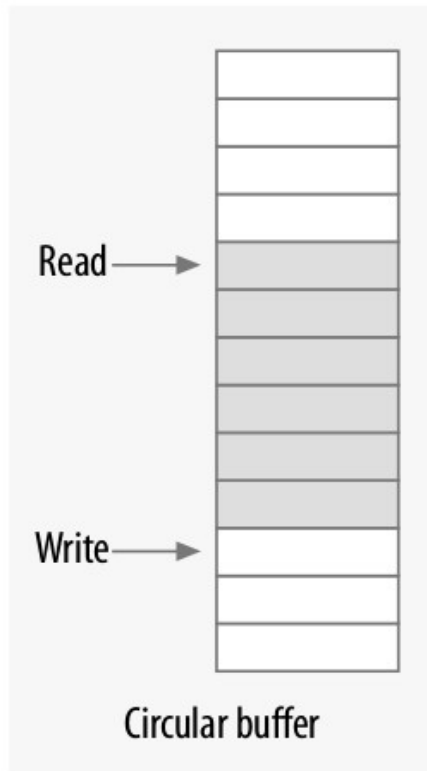
Example: network routing table is read/written using this mechanism

Completion Variable

- Easy way to synchronize between two tasks in the kernel when one task needs to signal to the other that an event has occurred.
- One task waits on the completion variable while another task performs some work.
- When the other task has completed the work, it uses the completion variable to wake up any waiting tasks.
- The idea is similar to semaphores. Completion variables merely provide a simple solution to a problem whose answer is otherwise semaphores.
- For example, the `vfork()` system call uses completion variables to wake up the parent process when the child process execs or exits.

Method	Description
<code>init_completion(struct completion *)</code>	Initializes the given dynamically created completion variable
<code>wait_for_completion(struct completion *)</code>	Waits for the given completion variable to be signaled
<code>complete(struct completion *)</code>	Signals any waiting tasks to wake up

Lock Free Mechanism



Synchronization Mechanisms

- Atomic integer and bit operation instructions
- Semaphores (counting or binary)
- Spinlocks
- Read/write semaphores/spinlocks
- Sequential lock
- Read copy update (RCU)
- Lock free mechanisms: no waiting
- Completion variables
- **Kernel preemption disallow**
- **Ordering and barriers**