

# Linux Device Drivers

---

## Agenda

- Multiple device instances
- Waiting Queue
- Synchronization

## Multiple device instances

- Refer slides and code

## Waiting Queue

- Each IO device and sync/ipc object has its own waiting queue.
- While a process is doing IO or waiting for sync/ipc, the process is in sleep state into the respective waiting queue. e.g. disk IO, printing on printer, blocking on semaphore/mutex, reading from pipe/message queue, etc.
- In Linux process sleep is of two types:
  - Interruptible sleep (S)
    - The sleeping process wake-up when event occurs (for which it was sleeping) or due to signal.
    - A process sleeping in Interruptible sleep can be woken up due to a signal that terminate the process or cause execution of the signal handler.
    - In such forcible wakeup, the system call (in which process was sleeping) will fail with errno=EINTR.
  - Uninterruptible sleep (D)
    - The sleeping process wake-up when event occurs (for which it was sleeping).
- In Linux device driver, when a process is to be blocked (until IO is performed on the device or some event occurs), the waiting queue can be used.
- Pseudo char device driver
  - Device --> struct kfifo buffer of size 32.
  - If a process is writing in the device, but device is full; then write operation fails with error -ENOSPC.
  - However this driver can be improved by blocking the process if device buffer is full (instead of failing).
- Use waiting queue in PCDD.

- In pchar\_device add waiting queue object.

```
wait_queue_head_t wr_wq;
```

- In module initialization, init waiting queue.

```
init_waitqueue_head(&wr_wq);
```

- In write() operation, wait if device buffer is full.

```
ret = wait_event_interruptible(wr_wq, !kfifo_is_full(&buffer));  
if(ret != 0) {  
    printk(KERN_INFO "%s: pchar_write() wakeup due to signal\n", THIS_MODULE->name);  
    return ret; //-ERESTARTSYS;  
}
```

- In read() operation after reading data from buffer,

```
wake_up_interruptible(&wr_wq);
```

- wait\_event\_interruptible()

```
while(1) {  
    if(condition)  
        break;  
    current process state = interruptible sleep  
    remove process from run queue
```

```
    add that process into the given waiting queue (blocked)
    if wakeup due to signal, return error.
}
```

- `wake_up_interruptible()`

```
remove the sleeping process from given waiting queue
change process state = runnable
add that process into the run queue
```

- Assignment: If buffer is empty, block the reader process until some data is written into the buffer.
  - step 1: create another waiting queue -- `rd_wq`.
  - step 2: initialize waiting queue in `module_init()`
  - step 3: In `pchar_read()`, `wait_event_interruptible()` -- `kfifo_is_empty()`
  - step 4: In `pchar_write()`, after write wakeup the reader process.

## Synchronization

- User-space programs are rarely multi-threaded. The kernel modules can be accessed by multiple user space applications concurrently. Hence kernel modules must be concurrency aware.
- If multiple applications access a device concurrently, then the device may malfunction or produce wrong results.
- To avoid this, the Synchronization should be implemented in the device drivers. The following Synchronization objects are present into Linux kernel.
  - Semaphore
  - Mutex (3.18+)
  - Spinlocks

## Semaphore

- Semaphore is internally a counter.
- `P(s)` -- wait op and `V(s)` -- signal op.
- Applications

- Counting
- Mutual exclusion
- Event (flag)
- Declared in `<linux/semaphore.h>`.

```
struct semaphore {  
    raw_spinlock_t    lock;  
    unsigned int      count;  
    struct list_head  wait_list;  
};
```

- Device driver should declare variable of this semaphore struct.
  - struct semaphore s;
- Semaphore Functions
  - void sema\_init(struct semaphore \*sem, int val);
    - e.g. sema\_init(&s, 3); // if initial count = 3
  - void up(struct semaphore \*sem);
    - up(&s); // increment sema count and wakeup a process if sleeping.
  - void down(struct semaphore \*sem);
    - down(&s); // decrement sema count and uninterruptible sleep if count is negative.
  - void down\_interruptible(struct semaphore \*sem);
    - down\_interruptible(&s); // decrement sema count and interruptible sleep if count is negative.

## Mutex

- Mutual exclusion
- Declared in `<linux/mutex.h>`.

```
struct mutex {  
    atomic_long_t    owner;
```

```
spinlock_t    wait_lock;  
struct list_head wait_list;  
};
```

- Device driver should declare variable of this mutex struct.
  - struct mutex m;
- Mutex functions
  - void mutex\_init(struct mutex \*m);
  - void mutex\_destroy(struct mutex \*m);
  - bool mutex\_is\_locked(struct mutex \*m);
    - returns true if mutex is already locked.
  - void mutex\_lock(struct mutex \*m);
    - lock the mutex if available.
    - if not available, block the current process in uninterruptible sleep.
  - int mutex\_lock\_interruptible(struct mutex \*m);
    - lock the mutex if available.
    - if not available, block the current process in interruptible sleep.
  - void mutex\_unlock(struct mutex \*lock);
    - release the mutex.
    - if any process is blocked for mutex, will be given access to it.
- Simple, straightforward mutexes with strict semantics:
  - Only one task can hold the mutex at a time.
  - Only the owner can unlock the mutex.
  - Multiple unlocks are not permitted.
  - Recursive locking is not permitted (i.e. a mutex cannot be locked multiple times).
  - A mutex object must be initialized via the API (Do not modify members directly).
  - A mutex object must not be initialized via memset or copying from other object.
  - Task may not exit with mutex held.
  - Memory areas where held locks reside must not be freed.
  - Held mutexes must not be reinitialized.
  - Mutexes may not be used in hardware or software interrupt (ISR) and Contexts such as tasklets and timers.

## Spinlock

- Spinlock is hardware/architecture based synchronization mechanism.
- Two processes cannot access spinlock simultaneously, in uni-processor or multi-processor environment.
- Semaphore/mutex should not be used in interrupt context (ISR), because ISR should never sleep.
- Semaphore is internally a counter and mutex is a lock. If multiple processes try to use the Semaphore/mutex simultaneously, there may be race condition for Semaphore/mutex itself.

### Solution 1

- When Semaphore count is incremented (V) or decremented (P), the processor interrupts can be disabled. This will ensure that no other process will preempt P and V operation, and thus no race condition for Semaphore.
- Semaphore P operation:
  - step 1: disable interrupts.
  - step 2: P operation (decrement and block if negative).
  - step 3: enable interrupts.
- Semaphore V operation:
  - step 1: disable interrupts.
  - step 2: V operation (increment and unblock if any process sleeping).
  - step 3: enable interrupts.
- This solution is applicable for uni-processor system. In multi-processor system, if interrupts are disabled, it will disable interrupts of current processor only. The process running on another processor can still access the Semaphore.
- Disabling (masking) interrupts also increases interrupt latencies.

### Solution 2

- When Semaphore count is incremented (V) or decremented (P), some hardware level synchronization mechanism should be used to access Semaphore by only one process.
- Spinlock is hardware level synchronization mechanism. Spinlocks are implemented using bus-holding instructions -- test\_and\_set() kind i.e. only one task can access the bus at a time. The test and set operations are done in same bus cycle i.e. bus remains locked until both operations are completed.
  - Example: ARM7 -- SWP, ARM Cortex -- LDREX, STREX.
- <https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Exclusive-accesses>

**Spinlock working**

- Spinlock is a variable -- 0 (unlocked/available) or 1 (locked/busy).
- Spinlock initialization. It is unlocked.

```
lock = 0;
```

- To lock a spinlock: If spinlock is busy, wait (busy wait loop); otherwise lock.

```
while(lock == 1)
    ;
lock = 1;
```

- To unlock a spinlock, clear it.

```
lock = 0;
```

**ARM7 SWP instruction**

- Spinlock implementation

```
spin:
    MOV r1, =lock
    MOV r2, #1

    SWP r3, r2, [r1] ; hold the bus until complete
```

```
CMP r3, #1
BEQ spin
```

- SWP instruction
  - SWP r3, r2, [r1]
    - r3 = \*r1 and \*r1 = r2;

#### ARM Cortex LDREX/STREX

- Refer: Joseph Yiu

#### Spinlock APIs

- `#include <linux/spinlock.h>`
- `spinlock_t lock;`
- `void spin_lock_init(spinlock_t *lock);`
- `void spin_lock(spinlock_t *lock);`
- `void spin_unlock(spinlock_t *lock);`

#### Spinlock in PCDD

- Only one process should write/read into device buffer.

```
spin_lock(&lock);
ret = kfifo_from_user(&dev->buffer, ubuf, len, &nbytes);
spin_unlock(&lock);
```

#### Semaphore vs Mutex vs Spinlock

- Spinlock are busy wait (not sleep).
- Can be used in interrupt context.



- Available only in kernel space.

### Spinlock usage

- Can be used only in kernel space.
- Can be used in process context or interrupt context.
- Use spinlock for minimal possible time duration.

```
spin_lock(&lock);  
// work  
spin_unlock(&lock);
```

- Do not sleep or exit with spinlock in lock state.

```
spin_lock(&lock);  
// sleep(5);           // DON'T DO THIS  
// mutex_lock(&m);      // DON'T DO THIS  
spin_unlock(&lock);
```

- Many times spinlock is used along with interrupt disable to avoid preemption.

```
spin_lock_irqsave(&lock, flags);  
// ...  
spin_unlock_irqrestore(&lock, flags);
```

### Linux execution contexts

- In Linux each code execute in one of the following context.
  - Process context --> Can be user process or kernel thread/daemon.

- Example 1: User program -- main() and lib fns execute in user process context.
- Example 2: Some kernel features like page stealing, ... execute in respective kernel thread context.
- Example 3: User program -- system call -- it is executed in user process context.
- The process context may sleep/block.
- The function activation records of invoked functions will be created on kernel stack of the current process.
- Interrupt context
  - Executing interrupt handler and ISR.
  - The interrupt context should never sleep/block.
  - For interrupt context, a separate kernel stack is created (per processor). It will have function activation record of ISR and functions called from it.