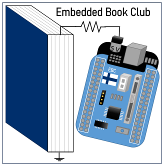


Linux Device Driver Development
Chapter 3: Dealing with Kernel Core Helpers

Jonathan Velasco

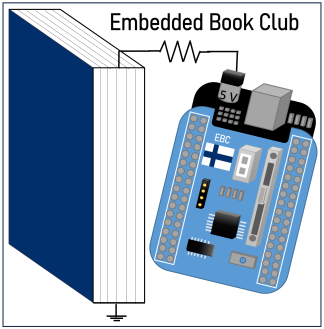
September 18th, 2023

Embedded Book Club Finland

We're knowledge sharing enthusiasts, focused on hosting in-person events, to bond over technical topics related to embedded systems.

Our goal is to

- Create a community focused on Embedded Systems and related topics
- Share knowledge, and to learn about new topics, trends and practices
- Having fun learning and working on projects together



Chapter 3 - Dealing with Kernel Core Helpers

The Kernel is a standalone piece of software that doesn't rely on the standard C library. This chapter covers topics related to the Kernel Core Helpers

- Locking mechanisms and shared resources
- Waiting, sleeping and delayed mechanisms
- Time management
- Implementing work-deferring mechanisms
- Kernel interrupt handling

Chapter 3 - Dealing with Kernel Core Helpers
Linux Kernel locking mechanisms and shared resources

Synchronization is needed in shared resources (memory location or peripheral devices) vs. contenders (processors, processes or threads).

This synchronization can be categorized as

- Locks: Exclusive ownership. Example: spinlocks and mutexes
- Conditional Variables: The kernel doesn't implement conditional variables, but instead provides the following mechanism:
  - Wait queue: waits for condition to be met, and works together with locks
  - Completion queue: Waits until task is completed. Mostly used in Direct Memory Access (DMA) transactions

These mechanisms are exposed by the kernel through Application Programming Interfaces (APIs).

Chapter 3 - Dealing with Kernel Core Helpers
Locks: Spinlocks

- Depend on hardware capabilities to provide atomic operations
- Lock **held by CPU**
- It disables the scheduler on the local CPU
- Task running on CPU cannot be preempted, except by interrupt requests IRQs
- Spinlocks are suitable for symmetrical multiprocessing (SMP) safety and excecuting atomic tasks
- Example: CPU_B is running and task B wants to acquire the spinlock, but the spinlock is being held by CPU_A running task. CPU_B spins in a while loop (blocking task B) until the lock is released.

Locks: Spinlocks API

Static definition macro: DEFINE_SPINLOCK
Dynamic runtime allocation: **spin_lock_init()**

- Use static definition whenever possible. Compile-time initialization and few lines of code.
- Definition in **include/linux/spinlock.h**
  - **spinlock_[un]lock(spinlock_t *lock)**
    - Enables/disables preemption
  - **spinlock_[un]lock_irq(spinlock_t *lock)**
    - Enables/disables preemtion and interrupts. Prevents CPU from being hogged by an interrupt.
  - **spinlock_[un]lock_irq[save/restore](spinlock_t *lock, unsigned long flags)**
    - Saves and restores IRQ status

Only disabling interrupts protects you from kernel preemption only in cases where the protected code does not trigger preemption itself (e.g., kernel functions that invoke the scheduler)

Chapter 3 - Dealing with Kernel Core Helpers
Locks: Mutexes

- Behaves like spinlock but code can sleep
- If lock is tried by a mutex held by another task, the tasks is suspended and woken up only when the mutex is released.
- No spinning. CPU can do something else.
- A mutex is a lock **held by a task**
- A mutex is a data structure that has a wait queue (puts contenders to sleep) and a spinlock to protect access to this wait queue:
- struct mutex {

```
    atomic_long_t owner; /* process that owns the lock */
    spinlock_t wait_lock; /* spinlock protecting wait_list */
    #ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq;
    #endif
    struct list_head wait_list; /* list of contenders put to sleep. keep
coherent with SMP systems */ };
```

Chapter 3 - Dealing with Kernel Core Helpers
Locks: Mutexes API

Static definition macro: DEFINE_MUTEX
Dynamic runtime allocation: **mutex_init()**

- Use static definition whenever possible. Compile-time initialization and few lines of code.
- Definition in **include/linux/mutex.h**
  - **int mutex_[un]lock(struct mutex lock);**
    - If unlocked, tasks sleeps immediately. Puts task in an uninterruptible state
  - **int mutex_[un]lock_interruptible(struct mutex lock);**
    - Sleep can be interrupted by any signal
  - **int mutex_[un]lock_killable(struct mutex lock);**
    - Can only interrupt sleep by signals that kill the task
  - They all return 0 if lock is succesfully acquired. iNTERRUPTIBLE VARIANTS RETURN -EINTR when locking attempt was interrupted by signal

Chapter 3 - Dealing with Kernel Core Helpers
Locks: Mutexes Rules

- Muxex is only held by one task at a time
- Can only be unlocked by owner
- Multiple, recursive, or nested locks/unlocks not allowed
- Mutex must be initialized via the API
- A task that holds a mutex may not exit, just as memory areas where held locks reside must not be freed
- Mutexes may not be used in hardware or software interrupt contexts such as tasklets and timers

Suitable use cases:

- Locking only in the user context
- If protected resources are not accessed from IRQ handler and the operations need not be atomic

For small critical sections spinlocks might be cheaper in terms of CPU cycles

Chapter 3 - Dealing with Kernel Core Helpers
Locks: Tryloc methods

- Use in cases where we might need to acquire the lock ONLY if not already held by another contender
- It checks the lock and without sleeping or spinning returns a status value: locked or not.
- mutexes and spinlocks provide trylock methods
  - mutex_trylock()
  - spin_trylock()

Chapter 3 - Dealing with Kernel Core Helpers
Kernel Waiting, sleeping and delay mechanisms

-

Chapter 3 - Dealing with Kernel Core Helpers
Kernel Waiting, sleeping and delay mechanisms

THIS SECTION IS INCOMPLETE