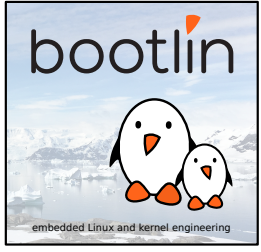
Concurrent Access to Resources: Locking

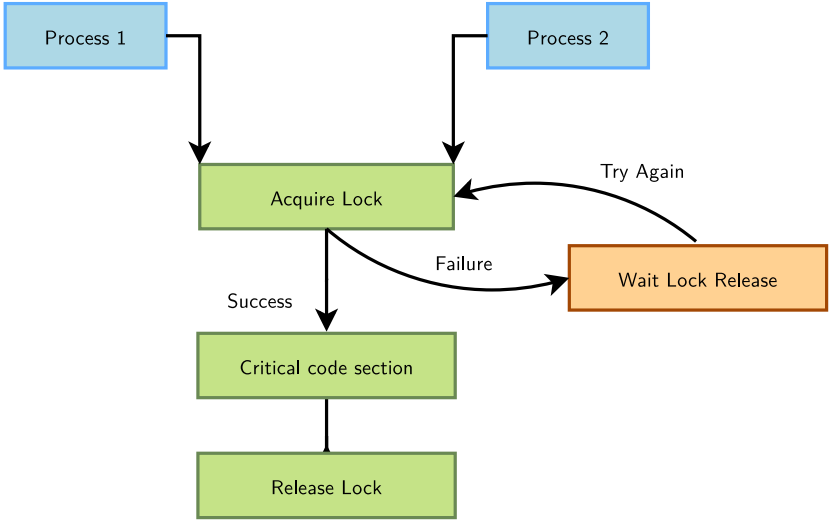


RED S
Reconfigurable & embedded Digital Systems

Sources of concurrency issues

- In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- Concurrency arises because of
  - *Interrupts*, which interrupts the current thread to execute an interrupt handler. They may be using shared resources (memory addresses, hardware registers...)
  - *Kernel preemption*, if enabled, causes the kernel to switch from the execution of one thread to another. They may be using shared resources.
  - *Multiprocessing*, in which case code is really executed in parallel on different processors, and they may be using shared resources as well.
- The solution is to keep as much local state as possible and for the shared resources that can't be made local (such as hardware ones), use locking.

Concurrency protection with locks



Linux mutexes

mutex = **mut***ual* **ex***clusion*

- The kernel's main locking primitive. It's a *binary lock*. Note that *counting locks* (*semaphores*) are also available, but used 30x less frequently.
- The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.
- Non-recursive only
- Mutex definition:
  - `#include <linux/mutex.h>`
- Initializing a mutex statically (unusual case):
  - `DEFINE_MUTEX(name);`
- Or initializing a mutex dynamically (the usual case, on a per-device basis):
  - `void mutex_init(struct mutex *lock);`

Locking and unlocking mutexes 1/2

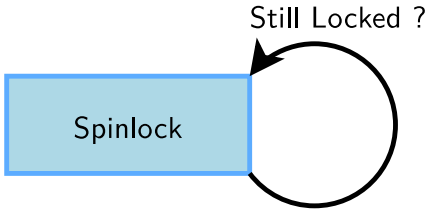
- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: can't be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable(struct mutex *lock);`
  - Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `int mutex_lock_interruptible(struct mutex *lock);`
  - Same, but can be interrupted by any signal.

Locking and unlocking mutexes 2/2

- `int mutex_trylock(struct mutex *lock);`
  - Never waits. Returns a non zero value if the mutex is not available.
- `int mutex_is_locked(struct mutex *lock);`
  - Just tells whether the mutex is locked or not.
- `void mutex_unlock(struct mutex *lock);`
  - Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

- Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.
- The critical section protected by a spinlock is not allowed to sleep.



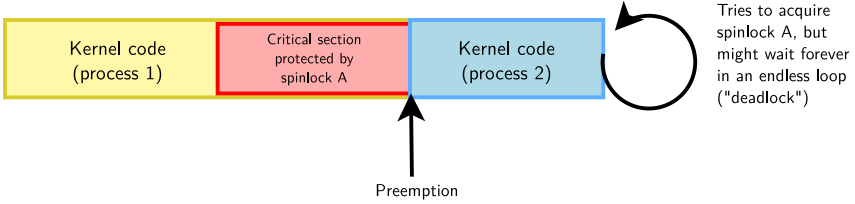
Initializing spinlocks

- Statically (unusual)
  - `DEFINE_SPINLOCK(my_lock);`
- Dynamically (the usual case, on a per-device basis)
  - `void spin_lock_init(spinlock_t *lock);`

Using spinlocks 1/3

Several variants, depending on where the spinlock is called:

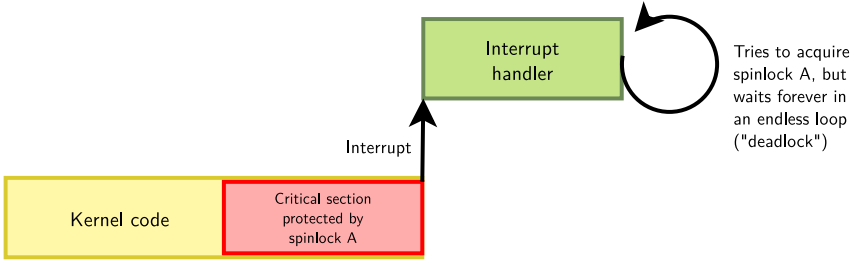
- `void spin_lock(spinlock_t *lock);`
- `void spin_unlock(spinlock_t *lock);`
  - Used for locking in process context (critical sections in which you do not want to sleep) as well as atomic sections.
  - Kernel preemption on the local CPU is disabled. We need to avoid deadlocks (and unbounded latencies) because of preemption from processes that want to get the same lock:



  - Disabling kernel preemption also disables migration to avoid the same kind of issue as pictured above from happening.

Using spinlocks 2/3

- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`
- `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);`
  - Disables / restores IRQs on the local CPU.
  - Typically used when the lock can be accessed in both process and interrupt context.
  - We need to avoid deadlocks because of interrupts that want to get the same lock.



Using spinlocks 3/3

- `void spin_lock_bh(spinlock_t *lock);`
- `void spin_unlock_bh(spinlock_t *lock);`
  - Disables software interrupts, but not hardware ones.
  - Useful to protect shared data accessed in process context and in a soft interrupt (*bottom half*).
  - No need to disable hardware interrupts in this case.
- Note that reader / writer spinlocks also exist, allowing for multiple simultaneous readers.

Spinlock example

- From drivers/tty/serial/uartlite.c
- Spinlock structure embedded into struct uart_port

```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

- Spinlock taken/released with protection against interrupts

```
static unsigned int ulite_tx_empty(struct uart_port *port) {
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        /* Do something */
        spin_unlock_irqrestore(&port->lock, flags);
}
```

Locking summary

Minimum locking requirements (see `kernel-hacking/locking`):

	IRQ Handler	**Softirq**	**Tasklet**	**User Context**
IRQ Handler	SLIS			
Softirq	SLI	SL		
Tasklet	SLI	SL	SL	
User Context	SLI	SLBH	SLBH	MLI

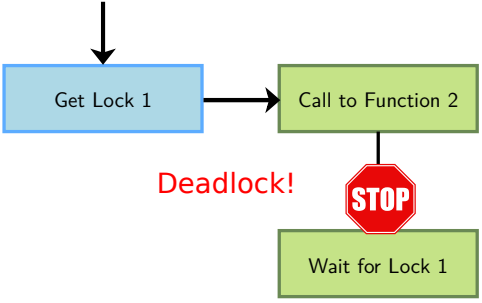
- SLIS: `spin_lock_irqsave()`
- SLI: `spin_lock_irq()`
- SL: `spin_lock()`
- SLBH: `spin_lock_bh()`
- MLI: `mutex_lock_interruptible()`

For spinlocks: you can always use `spin_lock_irqsave()` if not sure

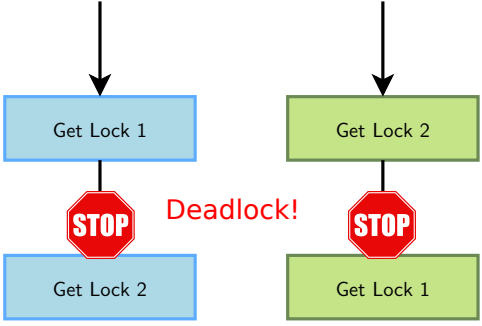
More deadlock situations

They can lock up your system. Make sure they never happen!

Rule 1: don't call a function that can try to get access to the same lock



Rule 2: if you need multiple locks, always acquire them in the same order!



Debugging locking

- Lock debugging: prove locking correctness
  - `CONFIG_PROVE_LOCKING`
  - Adds instrumentation to kernel locking code
  - Detect violations of locking rules during system life, such as:
    - Locks acquired in different order (keeps track of locking sequences and compares them).
    - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
  - Not suitable for production systems but acceptable overhead in development.
  - See `locking/lockdep-design` for details
- `CONFIG_DEBUG_ATOMIC_SLEEP` allows to detect code that incorrectly sleeps in atomic section (while holding lock typically).
  - Warning displayed in `dmesg` in case of such violation.

Concurrency issues

- Kernel Concurrency SANitizer framework
- CONFIG_KCSAN, introduced in Linux 5.8.
- Dynamic race detector relying on compile time instrumentation.
- Can find concurrency issues (mainly data races) in your system.
- See dev-tools/kcsan and https://lwn.net/Articles/816850/ for details.

Alternatives to locking

As we have just seen, locking can have a strong negative impact on system performance. In some situations, you could do without it.

- By using lock-free algorithms like *Read Copy Update* (RCU).
  - RCU API available in the kernel
  - See https://en.wikipedia.org/wiki/Read-copy-update for a coverage of how it works.
- When relevant, use atomic operations.

RCU API

- Conditions where RCU is useful:
  - Frequent reads but infrequent writes
  - Focus on getting consistent data rather than getting the latest data
- Kind of enforces ownership by enforcing space/time synchronization
- RCU API (`Documentation/RCU/whatisRCU.rst`):
  - `rcu_read_lock()` and `rcu_read_unlock()`: reclaim/release read access
  - `synchronize_rcu()`, `call_rcu()` or `kfree_rcu()`: wait for pre-existing readers
  - `rcu_assign_pointer()`: update RCU-protected pointer
  - `rcu_dereference()`: load RCU-protected pointer
- RCU mentorship session by Paul E. McKenney: `https://youtu.be/K-4TI5gFsig`

RCU example: ensuring consistent accesses (1/2)

Unsafe read/write

```
struct myconf { int a, b; } *shared_conf; /* initialized */

unsafe_get(int *cur_a, int *cur_b)
{
        *cur_a = shared_conf->a;
        /* What if *shared_conf gets updated now? The assignement is inconsistent! */
        *cur_b = shared_conf->b;
};

unsafe_set(int new_a, int new_b)
{
        shared_conf->a = new_a;
        shared_conf->b = new_b;
};
```

RCU example: ensuring consistent accesses (2/2)

Safe read/write with RCU

```
struct myconf { int a, b; } *shared_conf; /* initialized */

safe_get(int *cur_a, int *cur_b)
{
        struct myconf *temp;

        rcu_read_lock();
        temp = rcu_dereference(shared_conf);
        *cur_a = temp->a;
        /* If *shared_conf is updated, temp->a and temp->b will remain consistent! */
        *cur_b = temp->b;
        rcu_read_unlock();
};

safe_set(int new_a, int new_b)
{
        struct myconf *newconf = kmalloc(...);
        struct myconf *oldconf;

        oldconf = rcu_dereference(shared_conf);
        newconf->a = new_a;
        newconf->b = new_b;
        rcu_assign_pointer(shared_conf, newconf);
        /* Readers might still have a reference over the old struct here... */
        synchronize_rcu();
        /* ...but not here! No more readers of the old struct, kfree() is safe! */
        kfree(oldconf);
};
```

Atomic variables 1/2

```
#include <linux/atomic.h>
```

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Ideal for RMW (Read-Modify-Write) operations
- Main atomic operations on `atomic_t` (signed integer, at least 24 bits):
  - Set or read the counter:
    - `void atomic_set(atomic_t *v, int i);`
    - `int atomic_read(atomic_t *v);`
  - Operations without return value:
    - `void atomic_inc(atomic_t *v);`
    - `void atomic_dec(atomic_t *v);`
    - `void atomic_add(int i, atomic_t *v);`
    - `void atomic_sub(int i, atomic_t *v);`

Atomic variables 2/2

- Similar functions testing the result:
  - `int atomic_inc_and_test(...);`
  - `int atomic_dec_and_test(...);`
  - `int atomic_sub_and_test(...);`
- Functions returning the new value:
  - `int atomic_inc_return(...);`
  - `int atomic_dec_return(...);`
  - `int atomic_add_return(...);`
  - `int atomic_sub_return(...);`

Atomic bit operations

- Supply very fast, atomic operations
- On most platforms, apply to an `unsigned long *` type.
- Apply to a `void *` type on a few others.
- Ideal for bitmaps
- Set, clear, toggle a given bit:
  - `void set_bit(int nr, unsigned long *addr);`
  - `void clear_bit(int nr, unsigned long *addr);`
  - `void change_bit(int nr, unsigned long *addr);`
- Test bit value:
  - `int test_bit(int nr, unsigned long *addr);`
- Test and modify (return the previous value):
  - `int test_and_set_bit(...);`
  - `int test_and_clear_bit(...);`
  - `int test_and_change_bit(...);`

Kernel locking: summary and references

- Use mutexes in code that is allowed to sleep
- Use spinlocks in code that is not allowed to sleep (interrupts) or for which sleeping would be too costly (critical sections)
- Use atomic operations to protect integers or addresses

See `kernel-hacking/locking` in kernel documentation for many details about kernel locking mechanisms.

Further reading: see the classical *dining philosophers problem* for a nice illustration of synchronization and concurrency issues.

Image source: https://en.wikipedia.org/wiki/Dining_philosophers_problem)