On some platforms, so-called memory-mapped I/O is weakly ordered. On such platforms, driver writers are responsible for ensuring that I/O writes to memory-mapped addresses on their device arrive in the order intended. This is typically done by reading a 'safe' device or bridge register, causing the I/O chipset to flush pending writes to the device before any reads are posted. A driver would usually use this technique immediately prior to the exit of a critical section of code protected by spinlocks. This would ensure that subsequent writes to I/O space arrived only after all prior writes (much like a memory barrier op, mb(), only with respect to I/O).

A more concrete example from a hypothetical device driver:

```
        ...
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  val = readl(my_status);
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  val = readl(my_status);
CPU B:  ...
CPU B:  writel(newval2, ring_ptr);
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
```

In the case above, the device may receive newval2 before it receives newval, which could cause problems. Fixing it is easy enough though:

```
        ...
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  val = readl(my_status);
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  (void)readl(safe_register); /* maybe a config register? */
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  val = readl(my_status);
CPU B:  ...
CPU B:  writel(newval2, ring_ptr);
CPU B:  (void)readl(safe_register); /* maybe a config register? */
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

Here, the reads from safe_register will cause the I/O chipset to flush any pending writes before actually posting the read to the chipset, preventing possible data corruption.