1. 專案08\_spinlock:

Linux是一個多任務操作系統，肯定會存在多個任務共同操作同一段記憶體或者設備的情況，多個任務甚至中斷都能訪問的資源叫做共享資源。在驅動開發中要注意對共享資源的保護，也就是要處理對共享資源的並發訪問。

1. 併發與競爭:

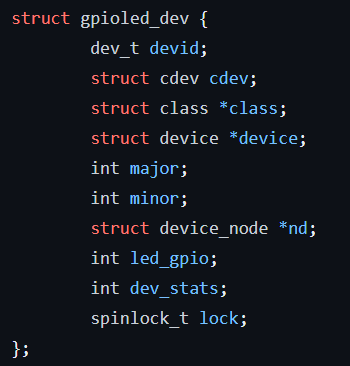
Linux系統是個多任務操作系統，會存在多個任務同時訪問同一片記憶體區域，這些任務可能會相互覆蓋這段記憶體中的數據，造成記憶體數據混亂。大致上原因有以幾項:

1. 多線程並發訪問，Linux是多任務（線程）的系統，所以多線程訪問是最基本的原因。
2. 搶佔式並發訪問，從2.6版本內核開始，Linux內核支持搶佔，也就是說調度程序可以在任意時刻搶佔正在運行的線程，從而運行其他的線程。
3. 中斷程序並發訪問，因為硬體中斷的權利是很大的。
4. SMP（多核）核間並發訪問，現在ARM架構的多核SOC很常見，多核CPU存在核間並發訪問。"
5. 保護的內容與方法:

我們實際要保護的內容就是共享數據，透故過防止併發的訪問來進行保護。

* 保護的方法有:
  + 原子操作(atomic):指的是不能再進一步分割的操作。而Linux Kernel中定義了atomic\_t的結構體於include/linux/types.h 文件中。(已於專案07\_atomic討論過)
    - 自旋鎖(Spinlock):
* 先前提到的原子操作(atomic)只支援整數類型的變數，但是在實際專案中不可能只需要保護整數類型的變數，因此需要有它方式來對數據進行保護。這裡的方法便是自旋鎖。
* 當一個線程要訪問某個共享資源的時候，首先要先獲取相應的鎖。鎖只能被一個線程持有，只要此線程不釋放持有的鎖，那麼其他的線程就不能獲取此鎖。對於自旋鎖而言，如果自旋鎖正在被線程 A 持有，線程 B 想要獲取自旋鎖，那麼線程 B 就會處於等待狀態。
* 這裡我們可以看到自旋鎖的一個缺點：那就是等待自旋鎖的線程會一直處於自旋狀態，這樣會浪費處理器時間，降低系統性能，所以自旋鎖的持有時間不能太長。自旋鎖適用於短時期的輕量級加鎖，如果遇到需要長時間持有鎖的場景，那就需要換其他的方法了
  + - 訊號量(Semaphore): 之後專案09\_semaphore會提到。
    - 互斥鎖(Mutex): 之後專案10\_mutex會提到。

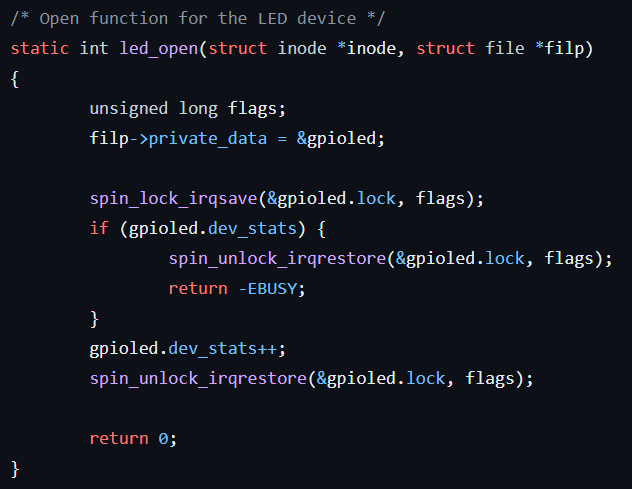
1. 實際spinlock()使用:
2. 首先需要把spinlock變數定義在設備結構體中。(如圖一)



(圖一: 設備結構體)

1. 在open()函式及release()函式中透過spin\_lock\_irqsave()保存中斷狀態並獲取spinlock，且透過spin\_unlock\_irqrestore()，恢復中斷狀態並釋放spinlock。

* 這裡補充一點，被自旋鎖保護的臨界區一定不能調用任何能夠引起睡眠和阻塞的API函數，否則的話會可能會導致死鎖現象的發生。自旋鎖會自動禁止搶佔，也就是說當線程A得到鎖以後會暫時禁止內核搶佔。如果線程A在持有鎖期間進入了休眠狀態，那麼線程A會自動放棄CPU使用權。線程B開始運行，線程B也想要獲取鎖，但是此時鎖被A線程持有，而且內核搶佔還被A禁止了！線程B無法被調度出去，那麼線程A就無法運行，鎖也就無法釋放，這樣，死鎖發生了！
* 此處spin\_lock\_irqsave()保存中斷狀態，禁止本地中斷並獲取spinlock，且透過spin\_unlock\_irqrestore()，恢復中斷狀態，啟用本地中斷並釋放spinlock。如此一來，當Thread獲取spinlock後就不會被搶占進入到睡眠或阻塞狀態了。(如下圖二，圖三)

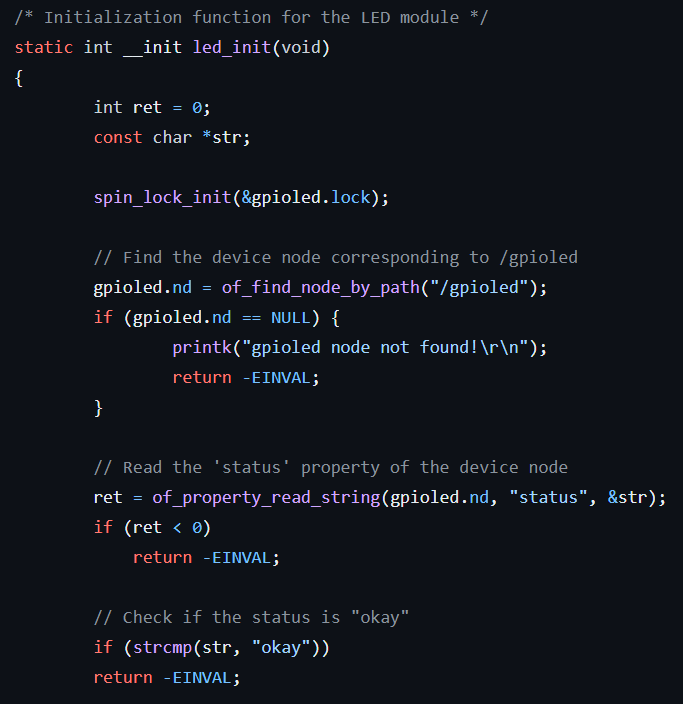


(圖二:open()函式內容)



(圖三:release()函式內容)

1. 最後在驅動入口函數init()函式中透過spin\_lock\_init()初始化spinlock就可以使用了。(如下圖四)



(圖四:驅動入口函數init()函式內容)

※總結:

原子操作(atomic)只支援整數類型的變數，但是在實際專案中不可能只需要保護整數類型的變數，因此需要其他方式進行共享資源的保護，例:spinlock。

而在使用spinlock需要注意死鎖現象發生。為防止死鎖現象可以透過spin\_lock\_irqsave()保存中斷狀態，禁止本地中斷並獲取spinlock，且透過spin\_unlock\_irqrestore()，恢復中斷狀態，啟用本地中斷並釋放spinlock。

如此一來，當Thread獲取spinlock後就不會被搶占進入到睡眠或阻塞狀態。了，死鎖的情況也就不會發生了。