L9 Kernel Synchronization

2019年11月15日 19:15

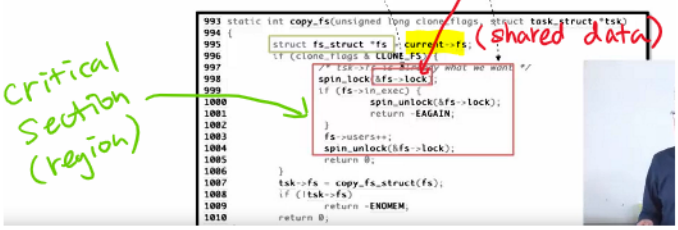
Kernel Synchronization

适用范围/情境.

Why sync? Overview

- Shared data
- Critical regions (critical sections)
  - Code paths that access/manipulate shared data

lock 放在 fs structure 里.



(shared data)

critical section (region)

critical region

操作 critical data (特指 foo 里 data 而不是 foo 程序)

Define "好"? → critical region 里不包含或者尽可能少地包含不操作 critical region foo 程序.

fs → 本身不是 shared data

current → fs 是 shared data

Overview (Cont.)

- Race condition
  - Two threads execute simultaneously within the same critical region
- Synchronization
  - Prevent unsafe concurrency and avid race conditions

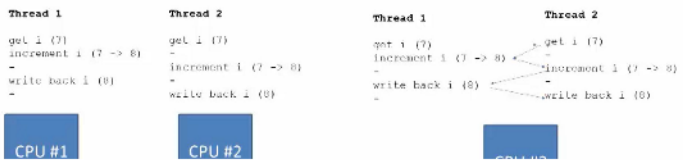
两个 thread 都在 critical section 里(操作 shared data) → race condition (不易重现 bug)

e.g. 两个 CPU 都在跑 Kernel 同一段操作 shared data 的 foo 程序

Example

- i: shared data
- i++

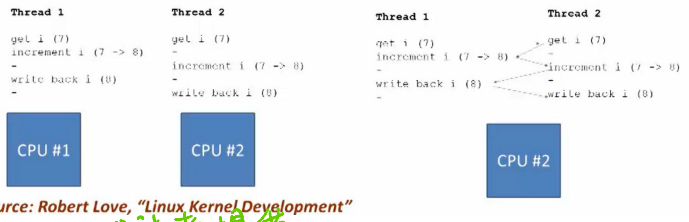
Thread 1	Thread 2
get i (7)	-
increment i (7 -> 8)	-
write back i (8)	-
-	get i (8)
-	increment i (8 -> 9)
-	write back i (9)



Example

- i: shared data
- i++

Thread 1	Thread 2
get i (7)	-
increment i (7 -> 8)	-
write back i (8)	-
-	get i (8)
-	increment i (8 -> 9)
-	write back i (9)



Source: Robert Love, "Linux Kernel Development"

用CPU的设计者提供

Solution

- Atomic instruction
  - X86 instruction, compare and exchange (CMPXCHG)

Thread 1	Thread 2
increment i(7 -> 8)	-
-	increment i (8 -> 9)

or

Thread 1	Thread 2
-	increment i (7 -> 8)
increment i(8 -> 9)	-

单核：OK!

单一变量原子存取

多核CPU必须保证

两个 atomic instruction 不会同时被执行.

缺点：

单一的变量用 atomic operation 解决就可以了.

但是大部分的 shared variable 都是一个数据结构

而不是一个变量.

Locking

Locking

- Using atomic instruction to handle shared variable is OK, how about data structure?

Thread 1	Thread 2
try to lock the queue	try to lock the queue
succeed: acquired lock	failed: waiting …
access queue …	waiting …
unlock the queue	waiting …
...	succeed: acquired lock
	access queue …
	unlock the queue
	...

Lock: 操作一段共享的数据的时候，哪怕我就是操作其中的一个变量我也要将他整个数据结构给加锁
一般的数据结构里面都包含了一个锁变量。

```
/* tsk->fs is already what we want */
spin_lock(&fs->lock);
if (fs->in_exec) {
        spin_unlock(&fs->lock);
        return -EAGAIN;
}
fs->users++;
spin_unlock(&fs->lock);
return 0;
```

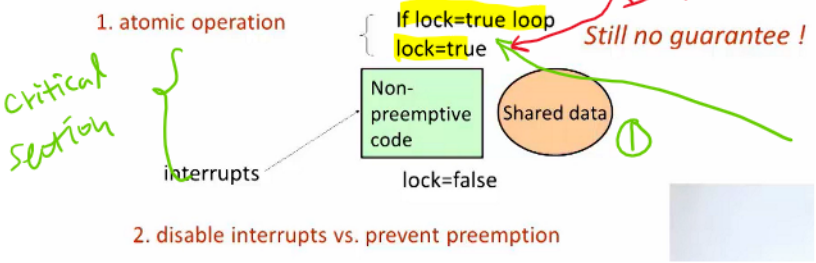
只是操作了 user，但也要 lock 整个数据结构.

Preemption Vs. thread safe.

Preemption thread safe. ← 由lock保护的.

Proc. thread safe. ← 由lock保护的code. 尽管不可重入，但还是thread safe

Locking



进到这里说明lock没有为true，也就是没有其它人lock它，那我就马上把lock改为true（acquire the lock）

如果线程A检查了lock，发现它还是为false的时候，还未将lock改为true，此时进行了上下文切换，线程B还是能进入critical section。
原因是 [redacted] 的部分还是多条指令。
修改办法： [redacted] 部分用atomic operation重写。

test and set
当一个人拿到lock是set lock false.

有2个问题：① ②

② 就算进到 critical section 了也不能保证不会出现 racing condition，因为这个 thread 可能会被 interrupt 打断，而且不能保证 interrupt subroutine 不会操作 shared data.

（如果能保证 interrupt subroutine 不会操作 shared data，那么只要在入口用 atomic operation 就够了）

interrupt subroutine 不操作 shared data，但引发了 context switch

e.g. timer 会定期打进来，进而执行 timer foo subroutine. 我们定义会 call scheduler，而无法保证其他 foo thread 会不会 call 进 kernel 对 shared data 进行操作.

（解决的办法是把 NEED_RESCHEDULE 去掉，kernel 在进到 interrupt subroutine foo 的时候会检查这个 flag，如果关了就不会 call Scheduler）

Causes of Concurrency

- Pseudo concurrency（单核）
  - Two threads do not actually execute simultaneously but interleave with each other (in critical region)
- True concurrency （多核）
  - Two threads execute simultaneously on two processors (SMP)

OS支持preemption：

OS支持在某一时刻突然停下去执行另一个（thread）foo 程序

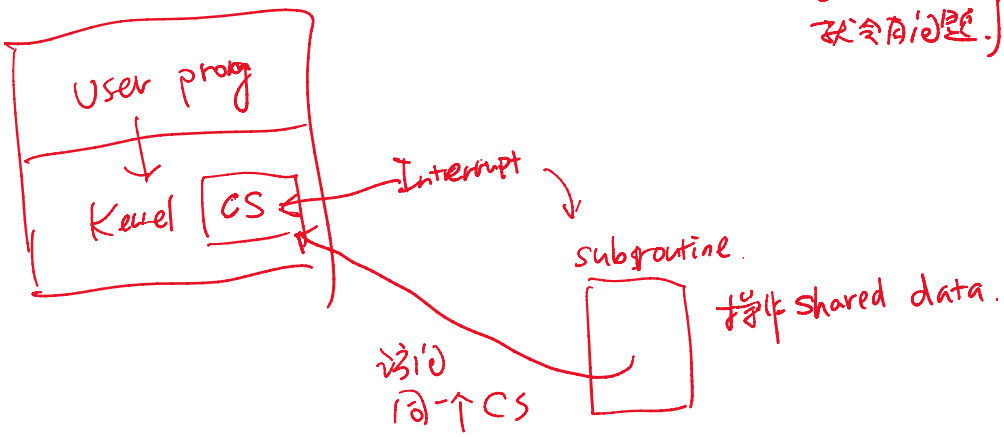
Causes of Concurrency (Cont.)

- Interrupts (Interrupt subroutine 操作 shared data)
  - An interrupt can occur asynchronously at almost any time, interrupting the currently executing code
- Softirqs and tasklets
  - The kernel can raise or schedule a softirq or tasklet at almost any time, interrupting the currently executing code
- Kernel preemption
  - Because the kernel is preemptive, one task in the kernel can preempt another (一个user操作kernel的时候出现preemption. 多个user trap进kernel操作同一块shared data)
- Sleeping and synchronization with user-space
  - A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process (hold resource的时候是不能sleep的. 因为会进行reschedule.)
- Symmetrical multiprocessing
  - Two or more processors can execute kernel code at exactly the same time

主动 被动

NEED_RESCHEDULE相关.

多个thread进入kernel如果对到同一块shared data 就会有问题.



Causes of Concurrency (Cont.)

- Interrupt-safe
  - Code that is safe from concurrent access from an interrupt handler
- SMP-safe
  - Code that is safe from concurrency on symmetrical multiprocessing machines
- preempt-safe
  - Code that is safe from concurrency with kernel preemption

← 不会出现 interrupt concurrent access shared variable的现象.

Kernel 支持 Interrupt ≠ Preempt

Preempt：一个进程在执行内核程序的时候，除非自愿放弃执行，否则另一个进程的用户态程序不能抢占执行。也就是说，内核是不可抢占或者说内核是不可重入的。

在kernel中工作很到一半,

e.g. 在proc A还没有lock任何shared resource的时候也可以被interrupt.

Preemption Kernel的优点：

实时性很高

Linux：早期仅支持Interrupt. 后期才支持preemption和SMP.

Protect the data structure!

Not the code!

因此不要Lock锁定的范围设定得过大.

BKL → 切切切 → 如今的Kernel不能被interrupt的区域已经很小了.

准时 Real Time ① 做事要多久
② CPU什么时候把事情交给你

(CPU什么时候能"打进来", Lock粒度越细. Interrupt越能打进来, Response Time越小. 实时性越高.)

Improve Respond Time:
把大的critical section
拆成一些较小的critical section

Interrupt打进来到运行的时间是在1ms以下, 可以称之为Real Time OS了.

经常使用Spin Lock会让Response Time变大.

经常被打断: 事情不能专心做 (实时性 and 性能的trade off)

Questions to ask before locking

绝大部分的Kernel data structure都不是global的.

- Is the data global? Can a thread of execution other than the current one access it? 是不是per CPU的?
- Is the data shared between process context and interrupt context? Is it shared between two different interrupt handlers?

between processes
between process and int.
between int hand

- If a process is preempted while accessing this data, can the newly scheduled process access the same data?
- Can the current process sleep (block) on anything? If it does, in what state does that leave any shared data?

如果Lock了一个资源之后去睡觉, 那可能会出问题, 因为Lock的资源别人可能要用!

△: 操作系统设计出这么多种Synchronization的机制的原因在于保护share data有不同的等级(不同的问题)

Deadlocks

[prevent or resolve]
↑ 一般OS采取的策略.

Thread 1	Thread 2
acquire lock A	acquire lock B
try to acquire lock B	try to acquire lock A
wait for lock B	wait for lock A

Sleep waiting or Busy waiting forever!

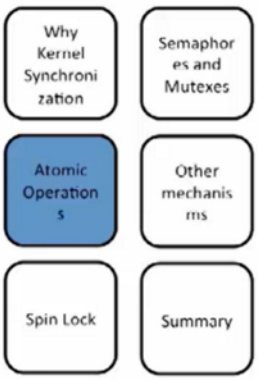
- Racing Condition
- Dead Lock.

How to prevent?

particular
request resource follows order and atomic

- Atomicity vs. ordering
- Make sure lock orders
  - Implement lock ordering
    - Nested locks must always be obtained in the same order
  - Do not double acquire the same lock
- The order of unlock does not matter with respect to deadlock

一般来说
Lock 顺序: A B C
解 Lock 顺序: C B A



Linux 同步机制之一:

Atomic Operations

—— 提供了一些不会被打断的指令

Atomic operation 是 Hardware dependent 的.
不同的 Hardware Platform 都有自己的实现公式
(不同的 Atomic 指令)

Atomic Operations

- Atomic integer operations (+ −)
- Atomic bitwise operations (0⇔1)

How to protect critical section?

- Atomic Integer Operations

ATOMIC_INIT(int i)	At declaration, initialize an atomic_t to i
int atomic_read(atomic_t *v)	Atomically read the integer value of v
void atomic_set(atomic_t *v, int i)	Atomically set v equal to i
void atomic_add(int i, atomic_t *v)	Atomically add i to v
void atomic_sub(int i, atomic_t *v)	Atomically subtract i from v
void atomic_inc(atomic_t *v)	Atomically add one to v
void atomic_dec(atomic_t *v)	Atomically subtract one from v
int atomic_sub_and_test(int i, atomic_t *v)	Atomically subtract i from v and return true if the result is zero; otherwise false
int atomic_add_negative(int i, atomic_t *v)	Atomically add i to v and return true if the result is negative; otherwise false
int atomic_dec_and_test(atomic_t *v)	Atomically decrement v by one and return true if zero; false otherwise
int atomic_inc_and_test(atomic_t *v)	Atomically increment v by one and return true if the result is zero; false otherwise

保护一个 integer 的变量.

Atomic operation

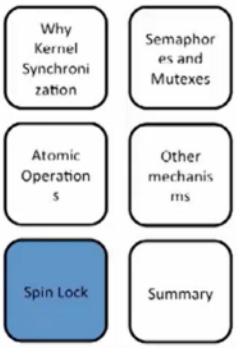
- Atomic Bitwise Operations

void set_bit(int nr, void *addr)	Atomically set the nr-th bit starting from addr
void clear_bit(int nr, void *addr)	Atomically clear the nr-th bit starting from addr
void change_bit(int nr, void *addr)	Atomically flip the value of the nr-th bit starting from addr
int test_and_set_bit(int nr, void *addr)	Atomically set the nr-th bit starting from addr and return the previous value
int test_and_clear_bit(int nr, void *addr)	Atomically clear the nr-th bit starting from addr and return the previous value
int test_and_change_bit(int nr, void *addr)	Atomically flip the nr-th bit starting from addr and return the previous value
int test_bit(int nr, void *addr)	Atomically return the value of the nr-th bit starting from addr

get.

Linux 里所有的保护机制都是基于 Atomic operation 来实现的.

e.g. Spin Lock 的实现.



Linux 同步机制之二:

Spin Lock

[适用范围]

— 短时间的 BusyWaiting. SMP.

Spin lock

- spin_lock
  - Normally, critical section is more than just a variable, spin lock protects a sequence of codes manipulating the critical section
  - Perform busy loops spins for waiting the lock to become available
  - Spin_lock disables the kernel preemption
  - On UP the locks compile away and do not exist; they simply act as markers to disable and enable kernel preemption
  - If kernel preempt is turned off, the locks compile away entirely

不希望有人将它打断. (因为 get the resource)

不希望又有人进来操纵这个变量

不会 reentry.

单核 Uni-processor

spin_lock implementation on SMP

关闭 preemption.

```
#define BUILD_LOCK_OPS(op, locktype)                                    \
void __lockfunc __raw_##op##_lock(locktype##_t *lock)                   \
{                                                                       \
        for (;;) {                                                      \
                preempt_disable();                                      \
                if (likely(do_raw_##op##_trylock(lock)))                \
                        break;                                          \
                preempt_enable();                                       \
                                                                        \
                if (!(lock)->break_lock)                                \
                        (lock)->break_lock = 1;                         \
                while (!raw_##op##_can_lock(lock) && (lock)->break_lock)\
                        arch_##op##_relax(&lock->raw_lock);             \
        }                                                               \
        (lock)->break_lock = 0;                                         \
}
```

try to lock (atomic operation)

没获得 Lock, 再把 preemption 打开, 再次 spin.

(说明为另一个 CPU 在 critical section 中)

为何是在等待为一个 thread 退出 critical section 的时候, 把此进程设为低功耗模式.

△ 注意, 如果为一个 CPU 永远不退出 CS. 那么执行 spin lock 的这颗 CPU 就"死掉"了

时候，把此进程放到低功耗模式下.

△ 注意，如果为个 CPU 永远不退出 CS，那么执行 spin lock 的这颗 CPU 就"死掉"了.

△：如果要 lock 很久，不建议使用 spin lock.

Spin lock

- Principles for using spin lock
  - Lock the data, not the code
  - Don't hold a spin lock for a long time (because someone may wait outside)
  - Don't recursively spin lock in Linux (like you lock your key in your car)
  - Spin lock (instead of using mutex) if you can finish within 2 context switches time
  - In a preemptive kernel, spin lock will perform preempt_disable( )
  - Can be used in interrupt handler (mutex/semaphores cannot) *must disable "local" interrupt*

别人等很久，那么这些时候与其做 spin，不如做些其他事情.
(等的人在 Busy waiting)

保证 local CPU 不会 interrupt

spin lock 里调 spin lock.
一定得到从而 spin，从而外面的 spin lock 永远都不能 unlock.

spin lock 是希望不进行 Context Switch 等到别人 release lock 后自己进入 CS.
而如果做的事情太长，完全可以先 Context switch 去做其他的事. 等做完了再 context switch 回来。(这样还不会浪费 CPU.)
如果等的时间短，Context switch 是不划算的. (Spin 开销 < Context Switch 开销)

☆. 不能被打断. 也可以按 interrupt.

(scheduler, Context Switch)

如果 data is shared by Process and interrupt handler.
则需要手动关闭 interrupt handler.

Spin lock

- When you use spin_lock(x) in the kernel, make sure you don't spend too much time in the lock, and there is no recursive spin_lock(x)
  - No spin_lock(x) in the spin_lock(x) period
- When you use spin_lock(x) in interrupt handler, make sure the interrupt has been disabled. Otherwise, use spin_lock_irq(x) or spin_lock_irqsave(x)

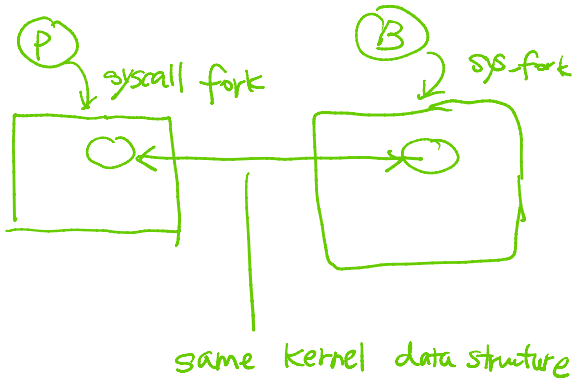
= spinlock + disable interrupt

加强版 spin_lock_irq( )

Considering following situations

1. You are writing kernel code (process context)
   - Shared data between processes
2. You are writing interrupt handler (interrupt context)
   - Shared data between interrupt handler
3. You are writing kernel code or interrupt handler
   - Shared data between interrupt handlers and processes

- *How could you use spin_lock to protect shared data?*



1. 可以
2. 基本的 spin_lock 不行. 需要加 disable interrupt.
3. 同 2.

Non-preemption

OK!

Interrupt

Not safe

Spin lock irq

- Disables interrupts and acquires the lock
- spin_unlock_irq() unlocks the given lock
- Disable interrupts (spin_lock_irq) and enable interrupts anyway (spin_lock_irqsave) no matter the status of current interrupts
  - Use of spin_lock_irq() therefore is not recommended

很少用. 因为无论原先 interrupt 是否开启. 都会进行 disable & enable.
如果原先 interrupt 是 disable 的. 那么行为就会有异常.

Spin lock irqsave

- Disables interrupt if it is not disabled and acquires the lock
- spin_unlock_irqrestore() unlocks the given lock and returns interrupts to their previous state

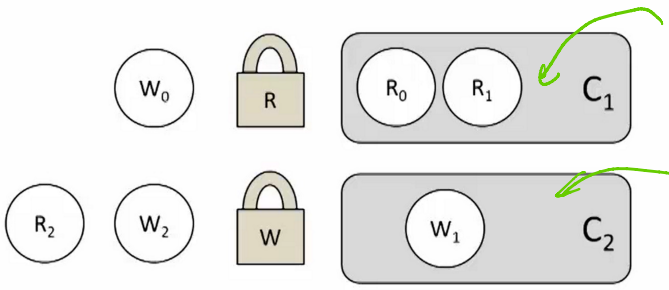
最好. (save and restore interrupt 状态)
但是保险.

Spin lock methods

spin_lock()	Acquires given lock
spin_lock_irq()	Disables local interrupts and acquires given lock
spin_lock_irqsave()	Saves current state of local interrupts, disables local interrupts, and acquires given lock
spin_unlock()	Releases given lock
spin_unlock_irq()	Releases given lock and enables local interrupts
spin_unlock_irqrestore()	Releases given lock and restores local interrupts to given previous state
spin_lock_init()	Dynamically initializes given spinlock_t
spin_trylock()	Tries to acquire given lock; if unavailable, returns nonzero
spin_is_locked()	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Kernel中foo DS有四是一个人写，有很多人可以读foo，这种用spin-lock不是很有效率.

Read/Write Spin Locks



Read foo critical section
可以有很多人进去 [提高Concurrency]

Write foo CS
互斥有一个人进

R_n – Reader kernel control path
W_n –Writer kernel control path
C_n – Critical region

Read write spin lock

- Lock usage can be clearly divided into readers and writers (e.g. search and update)
- Linux task list is protected by a reader-writer spin lock

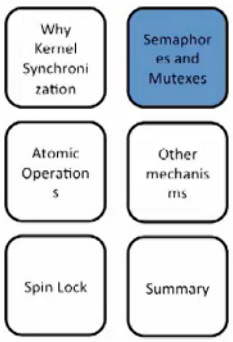
```
read_lock(&mr_rwlock);                  write_lock(&mr_rwlock);
/* critical section (read only) ... */  /* critical section (read and write) ... */
read_unlock(&mr_rwlock);                write_unlock(&mr_lock);
```

- Must split into reader/writer paths
  - You cannot do this

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

Read write spin lock

Method	Description
read_lock()	Acquires given lock for reading
read_lock_irq()	Disables local interrupts and acquires given lock for reading
read_lock_irqsave()	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
read_unlock()	Releases given lock for reading
read_unlock_irq()	Releases given lock and enables local interrupts
read_unlock_irqrestore()	Releases given lock and restores local interrupts to the given previous state
write_lock()	Acquires given lock for writing
write_lock_irq()	Disables local interrupts and acquires the given lock for writing
write_lock_irqsave()	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing
write_unlock()	Releases given lock
write_unlock_irq()	Releases given lock and enables local interrupts
write_unlock_irqrestore()	Releases given lock and restores local interrupts to given previous state
write_trylock()	Tries to acquire given lock for writing; if unavailable, returns nonzero
rw_lock_init()	Initializes given rwlock_t
rw_is_locked()	Returns nonzero if the given lock is currently acquired, or else it returns zero



Linux 同步机制之三：

Semaphores and Mutexes

—— 适用于要等很久的

- Mutex is a special case semaphore !

Semaphores

- sleeping locks —— 把自己process set to idle and call Scheduler.
- do not disable kernel preemption
- the mutex/semaphore place the task onto a wait queue and put the task to sleep
- well suited to locks that are held for a long time
- not optimal for locks that are held for very short periods
- can be obtained only in process context, **not interrupt context**
- can sleep while holding a semaphore
- **cannot hold a spin lock while you acquire a semaphore**
  - **How about hold a semaphore while you acquire a spin lock**

必须快速做完

(别在 interrupt 打不出来)

↳ 可以

因为 semaphore 会把 process put into sleep. 而此时由等待 hold 了一个 spin lock，

那么相当于等了 spin lock 并做了 sleep，这是绝对不行的！

反过来，已经 hold 了 semaphore，可以 hold spinlock。

Semaphores (Cont.)

- binary semaphore vs. counting semaphore (*not used much in the kernel*)
- down_interruptible()
  - attempts to acquire the given semaphore. If it fails, it sleeps in the TASK_INTERRUPTIBLE
- down()
  - If it fails, places the task in the TASK_UNINTERRUPTIBLE state if it sleeps (*normally not used*)

Semaphores

sema_init(struct semaphore *, int)	Initializes the dynamically created semaphore to the given count
init_MUTEX(struct semaphore *)	Initializes the dynamically created semaphore with a count of one
init_MUTEX_LOCKED(struct semaphore *)	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
down_interruptible(struct semaphore *)	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
down(struct semaphore *)	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
down_trylock(struct semaphore *)	Tries to acquire the given semaphore and immediately return nonzero if it is contended
up(struct semaphore *)	Releases the given semaphore and wakes a waiting task, if any

down(__)

主动返回

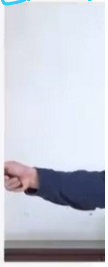
Implementation

```
1  void down(struct semaphore *sem)
2  {
3          unsigned long flags;
5          raw_spin_lock_irqsave(&sem->lock, flags);
6          if (likely(sem->count > 0))
7                  sem->count--;
8          else
9                  __down(sem);
10         raw_spin_unlock_irqrestore(&sem->lock, flags);
11 }
```

拿到控制sem的lock之后再对sem控制(spinlock)

```c
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
static inline int __sched __down_common(struct semaphore *sem, long st
                                        long tim
{
        struct task_struct *task = current;
        struct semaphore_waiter waiter;
        list_add_tail(&waiter.list, &sem->wait_list);
        waiter.task = task;
        waiter.up = 0;
        for (;;) {
                if (signal_pending_state(state, task))
                        goto interrupted;
                if (timeout <= 0)
                        goto timed_out;
                __set_task_state(task, state);
                raw_spin_unlock_irq(&sem->lock);
                timeout = schedule_timeout(timeout);
                raw_spin_lock_irq(&sem->lock);
                if (waiter.up)
                        return 0;
        }
 timed_out:
        list_del(&waiter.list);
        return -ETIME;
 interrupted:
        list_del(&waiter.list);
        return -EINTR;
}
```

TASK_UNINTERUPTIBLE 是 Idle mode，Scheduler 不会 schedule 到。
Linux 不会 schedule 到 Busy loop.



Mutex

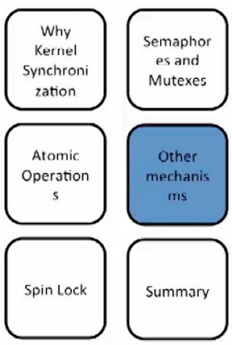
- Behaves similar to a semaphore with a count of one, but it has a simpler interface, more efficient performance

```
mutex_lock(&mutex);
/* critical region ... */
mutex_unlock(&mutex);
```

Method	Description
`mutex_lock(struct mutex *)`	Locks the given mutex; sleeps if the lock is unavailable
`mutex_unlock(struct mutex *)`	Unlocks the given mutex
`mutex_trylock(struct mutex *)`	Tries to acquire the given mutex; returns one if successful and the lock is acquired and zero otherwise
`mutex_is_locked (struct mutex *)`	Returns one if the lock is locked and zero otherwise

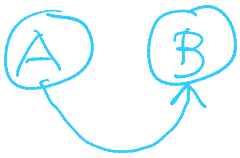
获得锁

判断.



Other Mechanisms

- Completion variables
  - an easy way to synchronize between two tasks in the kernel when one task needs to signal to the other that an event has occurred



- Sequential locks
  - shortened to seq lock, is a newer type of lock introduced in the 2.6 kernel. It provides a simple mechanism for reading and writing shared data

Other Mechanisms (Cont.)

- Interrupt disable
  - Allows a kernel control path to continue executing even when hardware issues IRQ signals
- Preemptions disable
  - For per-processor data access. If no spin locks are held, the kernel is preemptive, and it would be possible for a newly scheduled task to access this same variable

Other Mechanisms (Cont.)

- Memory barriers
  - May order

```
a = 1;
b = 2;
```

  - Will not reorder

```
a = 1;
b = a;
```

有dependency

  - Example, a=1, b=2, no mb(), rmb()

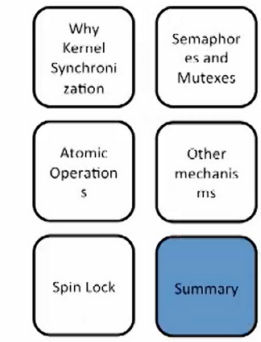
Read Memory Barrier

memory barrier

Thread 1	Thread 2
a = 3;	—
mb();	—
b = 4;	= b;
—	rmb();
—	d = a;

Kernel Synchro...

开始录音: 2019年11月24日 19:10



Protection of Shared Data in Kernel

Requirement	Recommended Lock
Low overhead locking	Spin lock
Short lock hold time	Spin lock
Long lock hold time	Mutex / Sem
Need to lock from interrupt context	Spin lock & disable interrupt
Need to sleep while holding lock	Mutex

Protection of Shared Data in Kernel

Kernel control paths accessing the data structure	UP protection	MP further protection
Exceptions (Syscall handler, pagefault handler)	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None or spin lock
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock

Will discuss deferrable functions (softirq and tasklet latter)

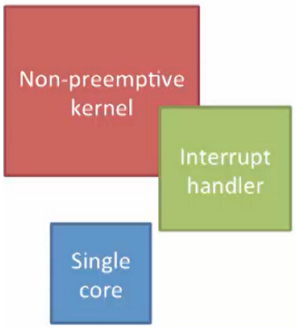
tasklet & Soft IRQ.

Protection of Shared Data in Kernel

- Exception handler
  - Such as system call, data can be shared by processes

Single Core + Non-preemptive kernel

Single Core + Non-preemptive kernel



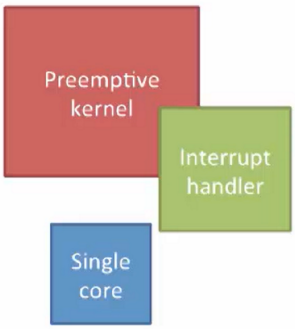
Non-preemptive kernel

Sync methods	Notes
local_irq_disable	Lock for short period
local_irq_save	Lock for short period, preferred
spin_lock	X
spin_lock_irq	=local_irq_disable
spin_lock_irqsave	=local_irq_save
Semaphore/mutex	Lock for long period

Interrupt handler

Sync methods	Notes
local_irq_disable	Lock for short period
local_irq_save	Lock for short period, preferred
spin_lock	X
spin_lock_irq	=local_irq_disable
spin_lock_irqsave	=local_irq_save
Semaphore/mutex	X

Single Core + Preemptive kernel



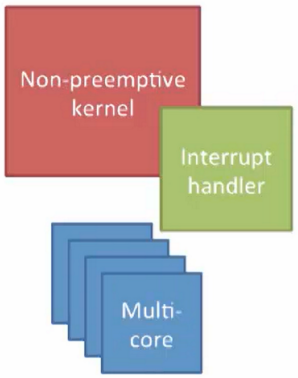
Preemptive kernel

Sync methods	Notes
local_irq_disable	OK for UP
local_irq_save	OK for UP (preferred)
spin_lock	OK, interrupt may occur
spin_lock_irq	OK, short period lock
spin_lock_irqsave	OK, short period lock (preferred)
Semaphore/mutex	OK, long period lock

Interrupt handler

Sync methods	Notes
local_irq_disable	OK for UP
local_irq_save	OK for UP (preferred)
spin_lock	OK, interrupt may occur
spin_lock_irq	OK, short period lock
spin_lock_irqsave	OK, short period lock (preferred)
Semaphore/mutex	No

Multi-Core + Non-preemptive kernel



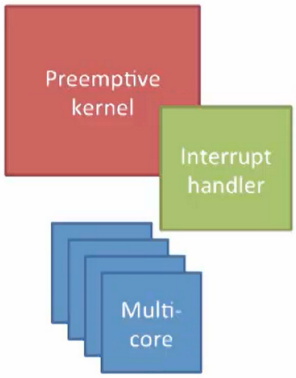
Non-preemptive kernel

Sync methods	Notes
local_irq_disable	
local_irq_save	
spin_lock	
spin_lock_irq	
spin_lock_irqsave	
Semaphore/mutex	

Interrupt handler

Sync methods	Notes
local_irq_disable	
local_irq_save	
spin_lock	
spin_lock_irq	
spin_lock_irqsave	
Semaphore/mutex	

Multi-Core + Preemptive kernel



Preemptive kernel

Sync methods	Notes
local_irq_disable	
local_irq_save	
spin_lock	
spin_lock_irq	
spin_lock_irqsave	
Semaphore/mutex	

Interrupt handler

Sync methods	Notes
local_irq_disable	
local_irq_save	
spin_lock	
spin_lock_irq	
spin_lock_irqsave	
Semaphore/mutex	