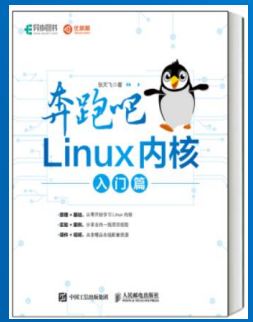
奔跑吧LINUX内核*入门篇

第九章 同步管理与锁机制

笨叔叔



目 录

- 同步与锁机制的基本概念
- 原子变量
- 内常屏障
- Spinlock
- 互斥体mutex
- 读写锁
- RCU
- 实验

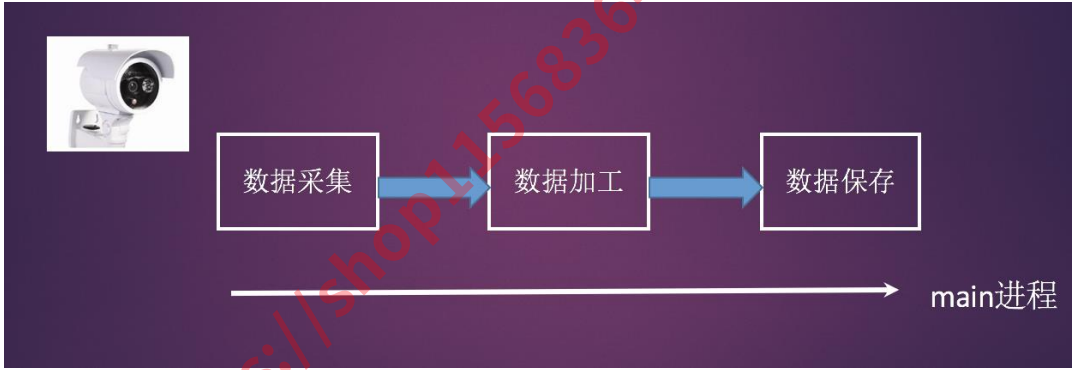
锁机制的基本概念

锁机制的基本概念

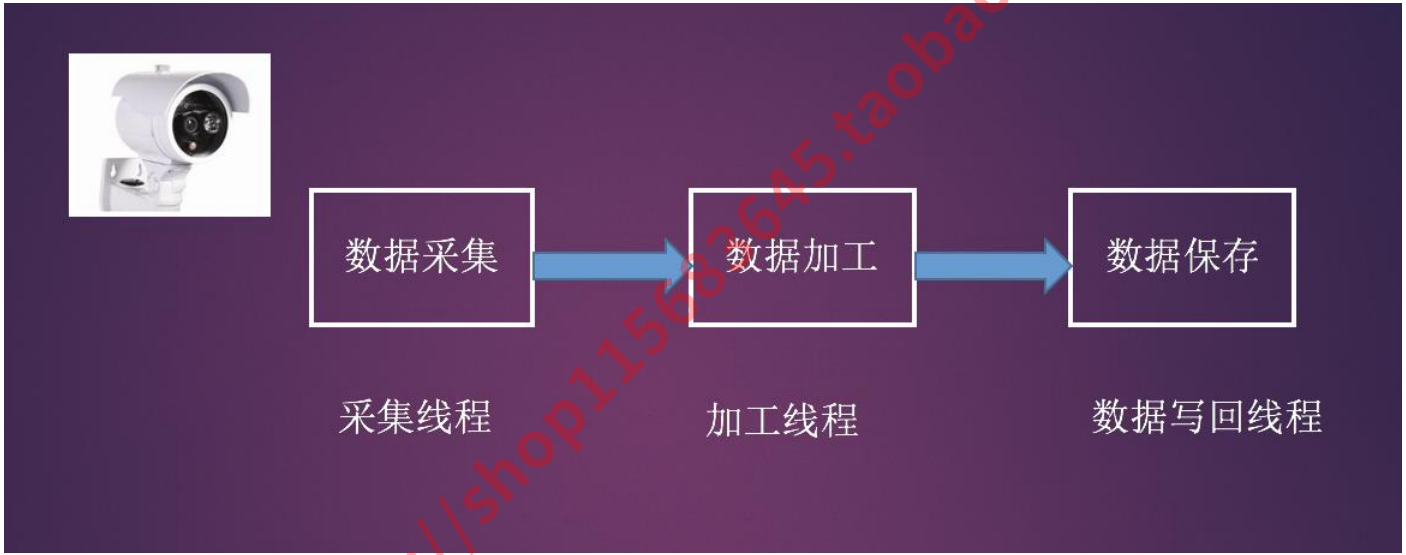
- 什么是顺序执行和并发执行?
- 为什么需要锁?
- 什么是互斥?
- 什么是临界区?
- 什么是生成者和消费者模型?

顺序执行

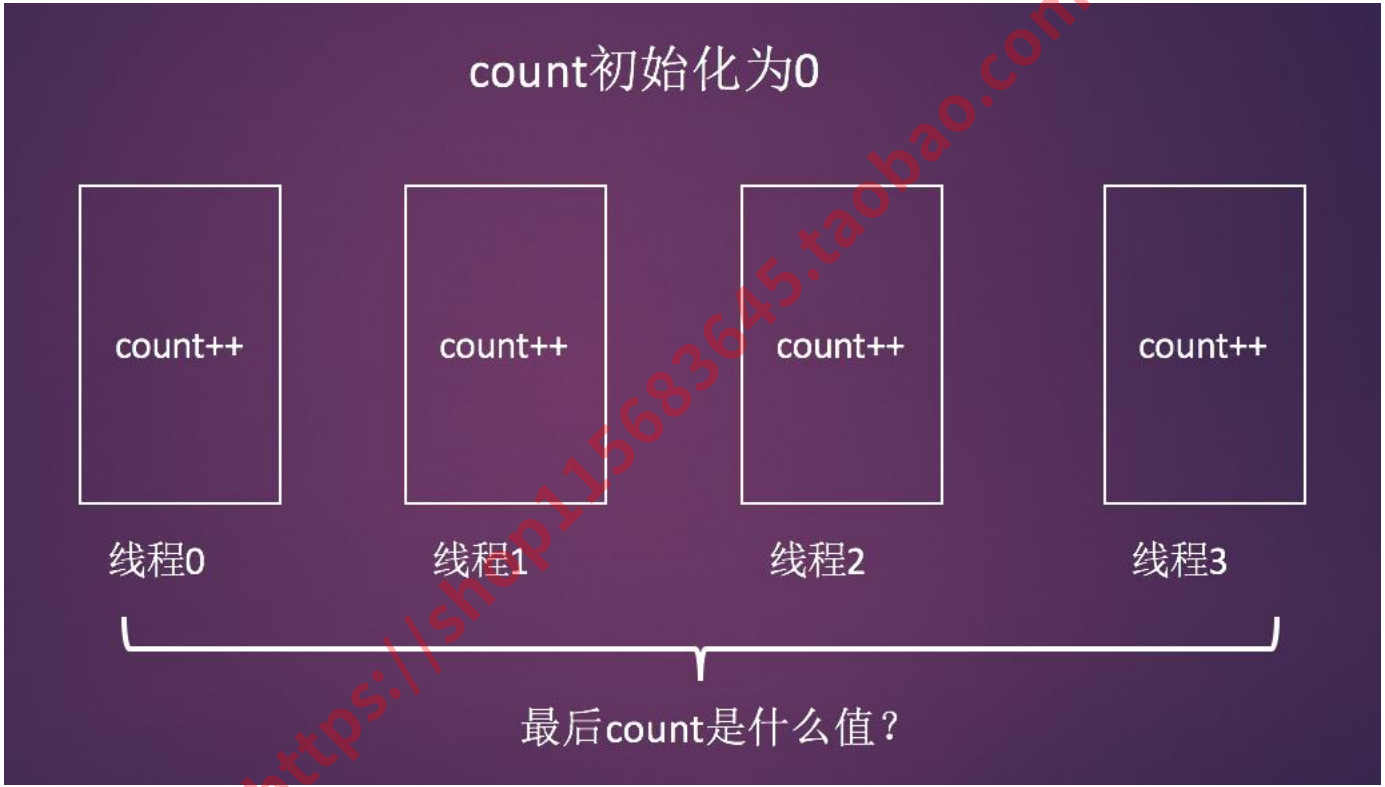
- 顺序执行就是程序执行是确定性的，也就是说指令按照程序顺序执行
- 比如单CPU的单进程程序



并发执行



并发执行例子

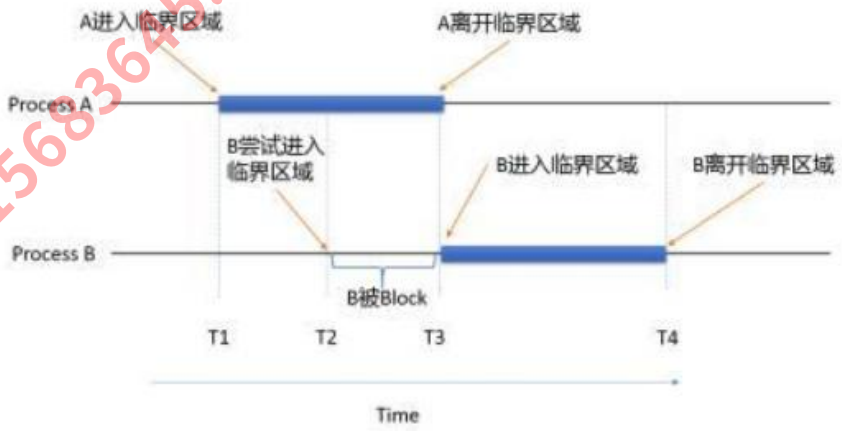


互斥（mutual exclusion）

- 在软件中保证一个时刻只有一个线程执行读写操作，那么这种在微观上确保一个时刻只允许一个线程执行特定的代码段的问题，称为互斥问题

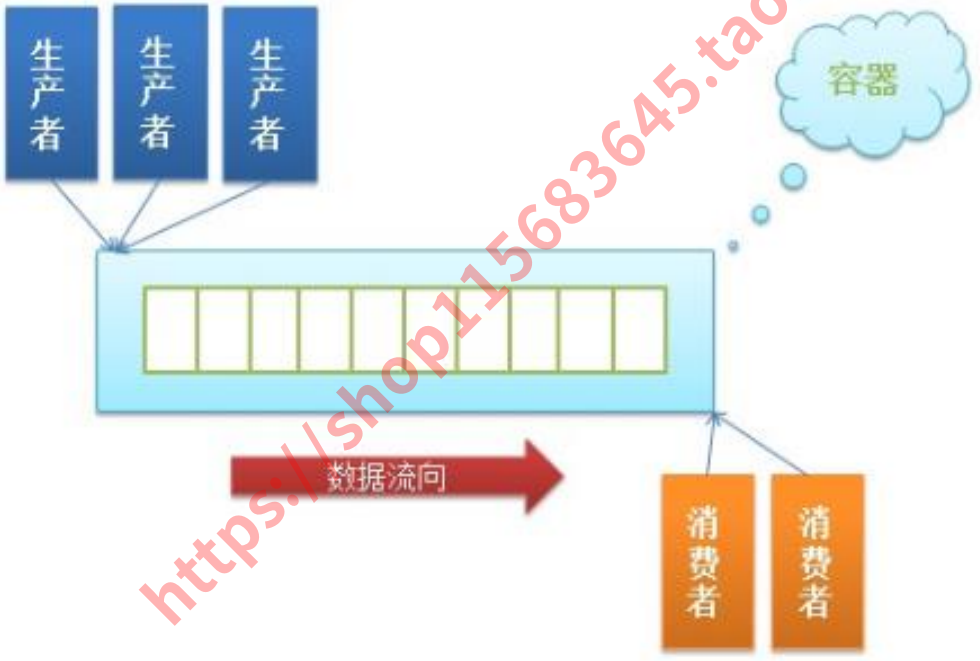
临界区

- 临界区：一个访问共用资源（例如：共用设备或是共用存储器）的程序片段，而这些共用资源又无法同时被多个线程访问的特性



生产者和消费者

- 经典的锁的机制



Linux内核编程中并发访问

- 单CPU以及单进程的系统中，会不会有并发访问?

- 单CPU多进程的系统中，有哪些因素会产生并发访问?
  - ✓ 中断处理程序可以打断软中断、tasklet和进程上下文的执行。
  - ✓ 软中断和tasklet之间不会并发，但是可以打断进程上下文的执行。
  - ✓ 在支持抢占的内核中，进程上下文之间会并发。
  - ✓ 在不支持抢占的内核中，进程上下文之间不会产生并发。

Linux内核编程中并发访问

- 在多CPU和多进程的系统里，有哪些因素会产生并发访问?

- 同一类型的中断处理程序不会并发，但是不同类型的中断有可能送达到不同的CPU上，因此不同类型的中断处理程序可能会存在并发执行。
- 同一类型的软中断会在不同的CPU上并发执行。
- 同一类型的tasklet是串行执行的，不会在多个CPU上并发。
- 不同CPU上的进程上下文会并发。

使用锁机制的要点

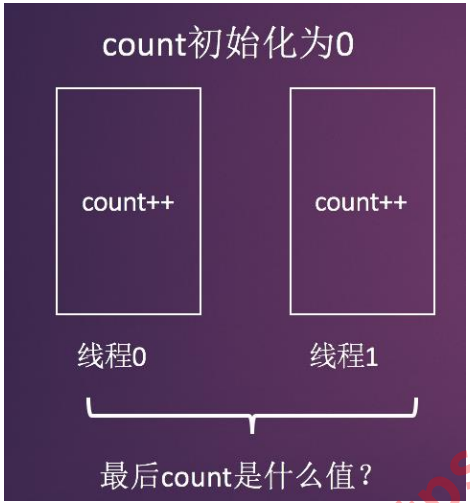
- **是保护资源或者数据，而不是保护代码**

- 除了当前内核代码路径外，是否还有其他内核代码路径会访问它？例如中断处理程序、工作者（worker）处理程序、tasklet处理程序、软中断处理程序等。
- 当前内核代码路径访问该资源数据时发生被抢占，被调度执行的进程会不会访问该数据？
- 进程会不会睡眠阻塞等待该资源？

原子变量

原子变量

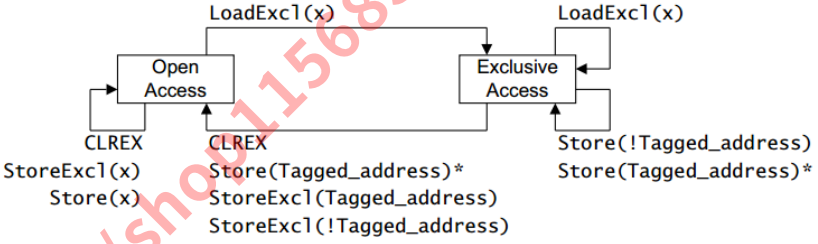
- 在体系结构中支持原子操作是所有操作系统锁机制的基础。
- 为什么要有原子变量呢?



CPU0	CPU1
thread_A_func	
load i= 0	thread_B_func Load i=0
i++	i++
store i (i=1)	store i (i=1)

ARM体系结构中的原子变量

- ARM中支持 独占访问的访问/存储指令
  - ✓ ldrex
  - ✓ strex
- 参考ARM v7手册《armv7_architecture_reference_manual》



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the tagged address to the most significant bits of the address x used for the operation.

Figure A3-3 Local monitor state machine diagram

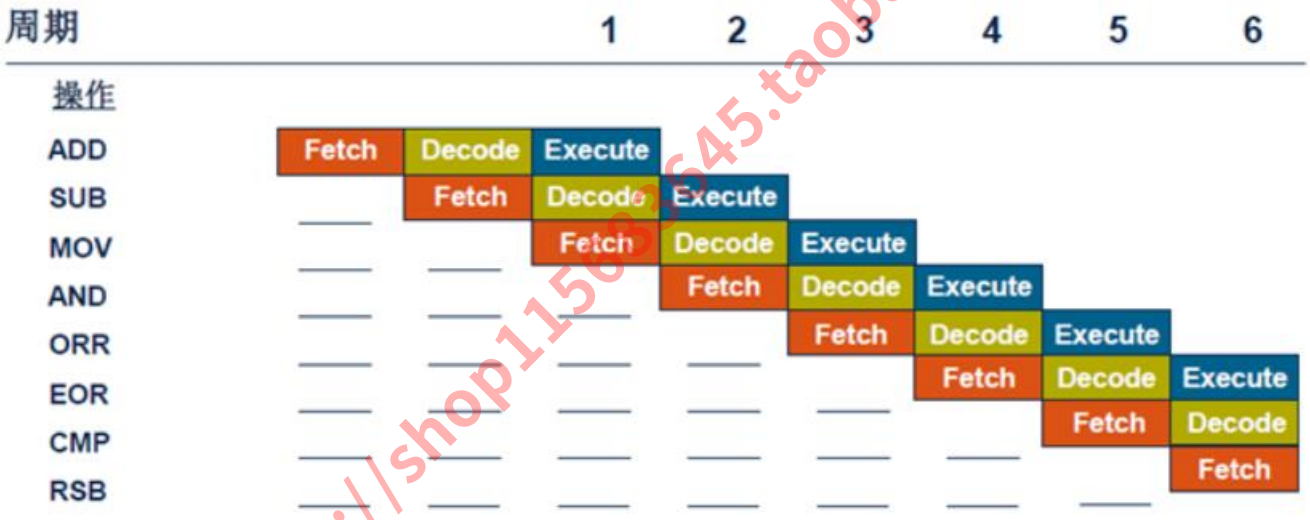
Linux内核实现的原子操作函数

- 头文件include/asm-generic/atomic.h，提供原子操作的函数

```
153 static inline void atomic_sub(int i, atomic_t *v)
154 {
155         atomic_sub_return(i, v);
156 }
157
158 static inline void atomic_inc(atomic_t *v)
159 {
160         atomic_add_return(1, v);
161 }
162
163 static inline void atomic_dec(atomic_t *v)
164 {
165         atomic_sub_return(1, v);
166 }
167
168 #define atomic_dec_return(v)            atomic_sub_return(1, (v))
169 #define atomic_inc_return(v)            atomic_add_return(1, (v))
170
171 #define atomic_sub_and_test(i, v)       (atomic_sub_return((i), (v)) == 0)
172 #define atomic_dec_and_test(v)          (atomic_dec_return(v) == 0)
173 #define atomic_inc_and_test(v)          (atomic_inc_return(v) == 0)
174
175 #define atomic_xchg(ptr, v)             (xchg(&(ptr)->counter, (v)))
176 #define atomic_cmpxchg(v, old, new)     (cmpxchg(&((v)->counter), (old), (new)))
```

内存屏障

经典流水线



内存屏障

- 为啥要有内存屏障?
- 如果没有内存屏障行不行?
- 在什么情况下的CPU不需要内存屏障，而在什么情况下的CPU又需要内存屏障?
- 那内存屏障产生的原因究竟是什么?
- ARM这个体系结构又是如何来解决这个问题?

内存屏障

- 顺序执行的处理器
- 乱序执行的处理器
  - ✓ 超标量技术
  - ✓ 乱序发射
  - ✓ 乱序执行

ARM的内存屏障指令

- 顺序执行的处理器
- 乱序执行的处理器
  - ✓ 超标量技术
  - ✓ 乱序发射
  - ✓ 乱序执行

ARM的内存屏障指令

- 数据存储屏障（Data Memory Barrier，DMB）
- 数据同步屏障（Data synchronization Barrier，DSB）
- 指令同步屏障（Instruction synchronization Barrier，ISB）

ARM内存屏障例子1

例 1：假设有两个 CPU 核 A 和 B，同时访问 Addr1 和 Addr2 地址。

```
Core A:
    STR R0, [Addr1]
    LDR R1, [Addr2]

Core B:
    STR R2, [Addr2]
    LDR R3, [Addr1]
```

ARM内存屏障例子2

例 2：假设 Core A 写入新数据到 Msg 地址，Core B 需要判断 flag 标志后才读入新数据。

```
Core A:
        STR R0, [Msg] @ 写新数据到Msg地址
        STR R1, [Flag] @ Flag标志新数据可以读

Core B:
    Poll_loop:
        LDR R1, [Flag]
        CMP R1,#0          @ 判断flag有没有置位
        BEQ Poll_loop
        LDR R0, [Msg]      @ 读取新数据
```

ARM内存屏障例子3

例3：在一个设备驱动中，写入一个命令到一个外设寄存器中，然后等待状态的变化。

```
STR R0, [Addr]          @ 写一个命令到外设寄存器
DSB
Poll_loop:
    LDR R1, [Flag]
    CMP R1,#0           @ 等待状态寄存器的变化
    BEQ Poll_loop
```

Linux中的内存屏障API

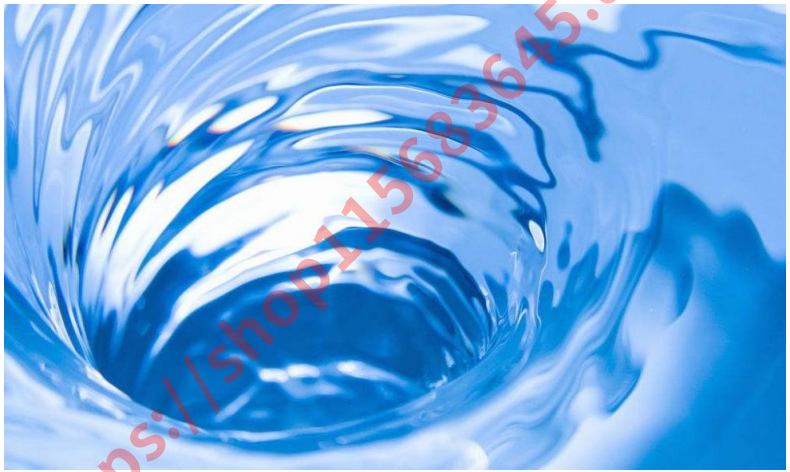
表 4.2　Linux 内核中的内存屏障函数接口

接口	描述
barrier()	编译优化屏障，阻止编译器为了性能优化而进行指令重排
mb()	内存屏障（包括读和写），用于SMP和UP
rmb()	读内存屏障，用于SMP和UP
wmb()	写内存屏障，用于SMP和UP
smp_mb()	用于SMP场合的内存屏障。对于UP不存在memory order的问题（对汇编指令），在UP上就是一个优化屏障，确保汇编和C代码的memory order一致
smp_rmb()	用于SMP场合的读内存屏障
smp_wmb()	用于SMP场合的写内存屏障
smp_read_barrier_depends()	读依赖屏障

spinlock

spinlock

- spin：旋转
- 原子变量能提供底层的锁机制，为啥还需要spinloc机制?



spinlock锁的基本概念

- 忙等待的锁机制。操作系统中锁的机制分为两类，一类是忙等待，另一类是睡眠等待。spinlock属于前者，当无法获取spinlock锁时会不断尝试，直到获取锁为止。
- 同一时刻只能有一个内核代码路径可以获得该锁。
- 要求spinlock锁持有者尽快完成临界区的执行任务。如果临界区执行时间过长，在锁外面忙等待的CPU比较浪费，特别是spinlock临界区里不能睡眠。
- spinlock锁可以在中断上下文中使用。

spinlock锁在Linux 4.0的定义

```
20 typedef struct raw_spinlock {
21         arch_spinlock_t raw_lock;
22 #ifdef CONFIG_GENERIC_LOCKBREAK
23         unsigned int break_lock;
24 #endif
25 #ifdef CONFIG_DEBUG_SPINLOCK
26         unsigned int magic, owner_cpu;
27         void *owner;
28 #endif
29 #ifdef CONFIG_DEBUG_LOCK_ALLOC
30         struct lockdep_map dep_map;
31 #endif
32 } raw_spinlock_t;
```

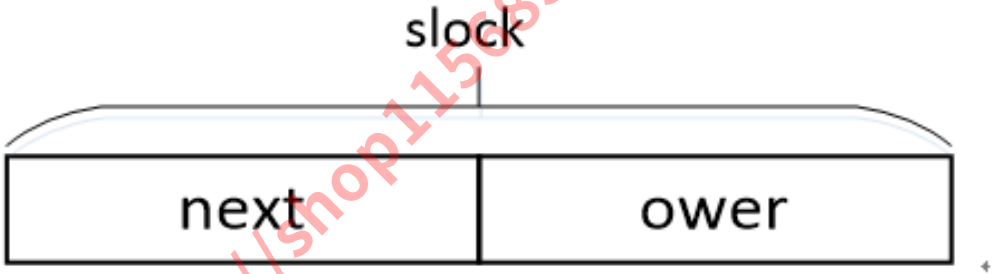
include/linux/spinlock_types.h

```
10 typedef struct {
11         union {
12                 u32 slock;
13                 struct __raw_tickets {
14 #ifdef __ARMEB__
15                         u16 next;
16                         u16 owner;
17 #else
18                         u16 owner;
19                         u16 next;
20 #endif
21                 } tickets;
22         };
23 } arch_spinlock_t;
24
```

arch/arm/include/asm/spinlock_types.h

ticket-based的spinlock

- spinlock的问题：在多核处理器中，spinlock锁的争用很激烈。
- 当该锁释放时，事实上有可能刚刚释放该锁的CPU马上又获得了该锁的使用权，或者说在同一个NUMA节点上的CPU都有可能抢先获取了该锁，而没有考虑那些已经在锁外面等待了很久的CPU



生活例子 – 上海饭店

- 假设某个饭店只有一张饭桌，刚开市时，next和owner都是0。
  - ✓ 第一个客户A来时，饭馆还没有顾客，所以客户A的等号牌是0，直接进餐，这时next++。
  - ✓ 第二个客户B来时，因为next为1，owner为0，说明锁被人持有。这时服务员给他1号的等号牌，让他在饭店门口等待，next++。
  - ✓ 第三个客户C来了，因为next为2，owner为0，服务员给他2号的等号牌，让他在饭店门口排队等待，next++。
  - ✓ 这时第一个客户A吃完埋单了，owner++，owner的值变为1。服务员会让等号牌和owner值相等的客户就餐，客户B的等号牌是1，所以现在客户B就餐。有新客户来时next++，服务员分配等号牌；客户埋单时owner++，服务员叫号，owner值和等号牌相等的客户就餐。

spinlock变种

- spin_lock_irq()主要防止本地中断处理程序和持有锁者之间存在锁的争用
- spin_lock_irqsave()函数会保存本地CPU当前的irq状态并且关闭本地CPU中断，然后获取自旋锁
- spin_lock_bh()函数用于处理进程和延迟处理机制导致的并发访问的互斥问题。

spinlock和raw_spin_lock

- 为什么有的代码用spin_lock()，而有的代码使用raw_spin_lock()?
- 实时补丁RT-patch，spinlock变成可抢占和睡眠的锁
- 使用方法：在绝对不允许被抢占和睡眠的临界区，应该使用raw_spin_lock，否则使用spinlock

信号量

信号量

- 信号量中最经典的例子莫过于生产者和消费者问题，它是一个操作系统发展历史上最经典的进程同步问题，最早由Dijkstra提出。

信号量API

- 信号量的定义

```
[include/linux/semaphore.h]

struct semaphore {
    raw_spinlock_t          lock;
    unsigned int            count;
    struct list_head        wait_list;
};
```

- down操作API函数

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_killable(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
int down_timeout(struct semaphore *sem, long jiffies);
```

- Up操作API函数

```
void up(struct semaphore *sem)
```

互斥体mutex

mutex

- 为啥需要有mutex这样一个互斥体?
- Mutex的语义相对于信号量要简单轻便一些，在锁争用激烈的测试场景下，Mutex比信号量执行速度更快，可扩展性更好

```
/*
 * Simple, straightforward mutexes with strict semantics:
 *
 * - only one task can hold the mutex at a time
 * - only the owner can unlock the mutex
 * - multiple unlocks are not permitted
 * - recursive locking is not permitted
 * - a mutex object must be initialized via the API
 * - a mutex object must not be initialized via memset or copying
 * - task may not exit with mutex held
 * - memory areas where held locks reside must not be freed
 * - held mutexes must not be reinitialized
 * - mutexes may not be used in hardware or software interrupt
 *   contexts such as tasklets and timers
 *
```

include/linux/mutex.h

Mutex特点

- 同一时刻只有一个线程可以持有Mutex。
- 只有锁持有者可以解锁。不能在一个进程中持有 Mutex，而在另外一个进程中释放它。因此Mutex不适合内核同用户空间复杂的同步场景，信号量和读写信号量比较适合。
- 不允许递归地加锁和解锁。
- 当进程持有Mutex时，进程不可以退出。
- Mutex必须使用官方API来初始化。
- Mutex可以睡眠，所以不允许在中断处理程序或者中断下半部中使用，例如tasklet、定时器等。

Mutex定义和API

- Mutex数据结构的定义

```
[include/linux/mutex.h]

struct mutex {
     atomic_t      count;
     spinlock_t        wait_lock;
     struct list_head    wait_list;
#if defined(CONFIG_MUTEX_SPIN_ON_OWNER)
     struct task_struct *owner;
#endif
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
     struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
};
```

- Mutex接口函数

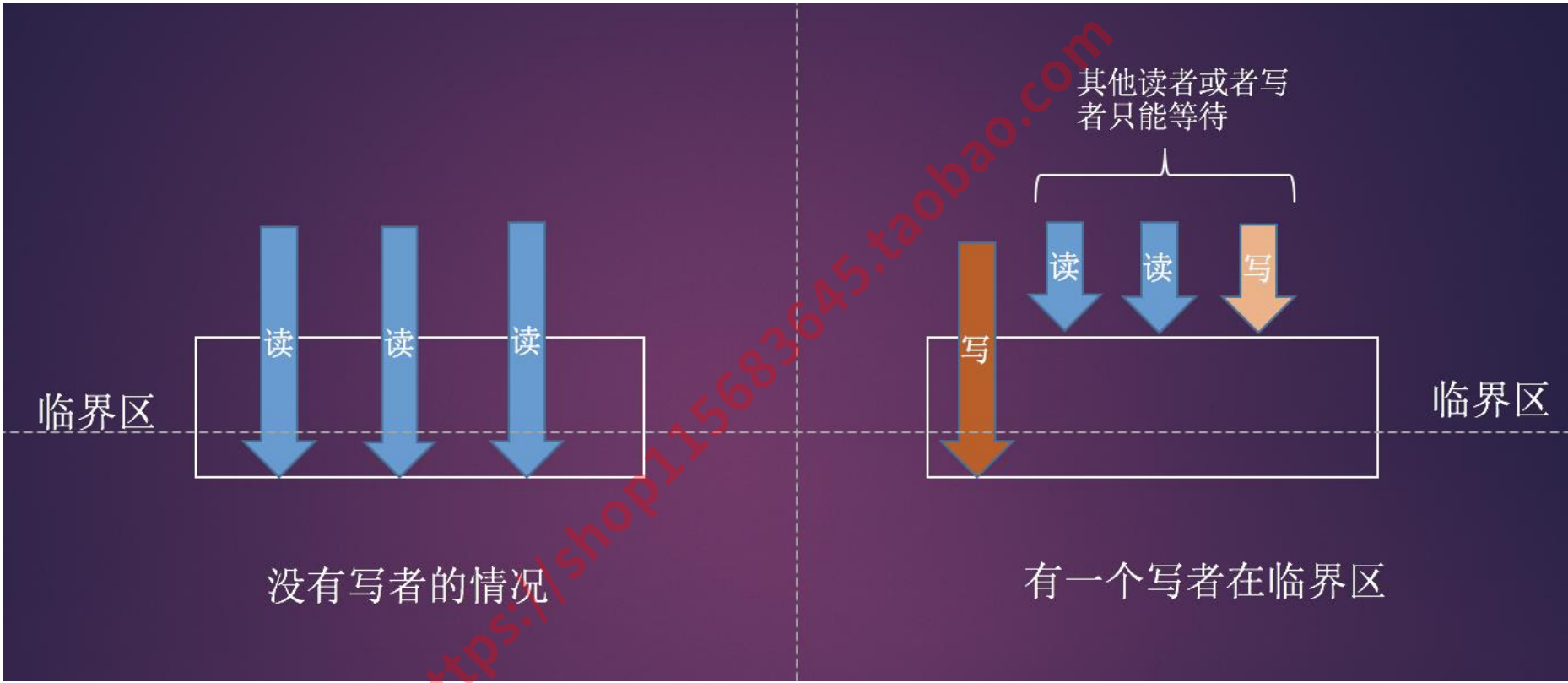
```
[include/linux/mutex.h]

#define DEFINE_MUTEX(mutexname) \
    struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
```

```
void __sched mutex_lock(struct mutex *lock)
void __sched mutex_unlock(struct mutex *lock)
```

读写锁

读写锁

- 信号量有一个明显的缺点—没有区分临界区的读写属性
- 读写锁通常允许多个线程并发地读访问临界区，但是写访问只限制于一个线程。
- 特点：
  - ✓ 允许多个读者同时进入临界区，但同一时刻写者不能进入。
  - ✓ 同一时刻只允许一个写者进入临界区。
  - ✓ 读者和写者不能同时进入临界区。



读写spinlock

- 读写spinlock数据结构的定义

```
[include/linux/rwlock_types.h]

typedef struct {
    arch_rwlock_t raw_lock;
} rwlock_t;

[arch/arm/include/asm/spinlock_types.h]
typedef struct {
    u32 lock;
} arch_rwlock_t;
```

- API接口：

```
[include/linux/rwlock.h]

rwlock_init() 初始化rwlock
write_lock()  申请写者锁
write_unlock() 释放写者锁
read_lock() 申请读者锁
read_unlock() 释放读者锁
read_lock_irq() 关闭中断并且申请读者锁
write_lock_irq() 关闭中断并且申请写者锁
write_unlock_irq() 打开中断并且释放写者锁
…
```

读写信号量

- 读写信号量数据结构的定义

```
[include/linux/rwsem.h]

struct rw_semaphore {
     long count;
     struct list_head wait_list;
     raw_spinlock_t wait_lock;
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
     struct optimistic_spin_queue osq; /* MCS锁 */
     struct task_struct *owner;
#endif
};
```

- API接口：

```
init_rwsem(struct rw_semaphore *sem);
void __sched down_read(struct rw_semaphore *sem)
void up_read(struct rw_semaphore *sem)
void __sched down_write(struct rw_semaphore *sem)
void up_write(struct rw_semaphore *sem)
int down_read_trylock(struct rw_semaphore *sem)
int down_write_trylock(struct rw_semaphore *sem)
```

RCU

RCU

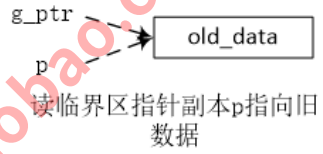
- RCU - read-copy-update
- Linux内核中已经有了原子操作、spinlock、读写spinlock、读写信号量、mutex等锁机制，为什么要单独设计一个比它们实现要复杂得多的新机制呢？
- RCU机制要实现的目标是，希望读者线程没有同步开销，或者说同步开销变得很小，甚至可以忽略不计，不需要额外的锁，不需要使用原子操作指令和内存屏障，即可畅通无阻地访问；而把需要同步的任务交给写者线程，写者线程等待所有读者线程完成后才会把旧数据销毁

RCU常用的接口API

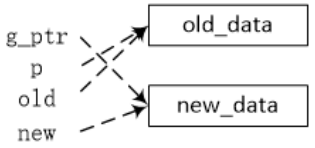
- rcu_read_lock()/ rcu_read_unlock()：组成一个RCU读临界。
- rcu_dereference()：用于获取被RCU保护的指针（RCU protected pointer），读者线程要访问RCU保护的共享数据，需要使用该函数创建一个新指针，并且指向RCU被保护的指针。
- rcu_assign_pointer()：通常用在写者线程。在写者线程完成新数据的修改后，调用该接口可以让被 RCU 保护的指针指向新创建的数据，用 RCU 的术语是发布（Publish）了更新后的数据。
- synchronize_rcu()：同步等待所有现存的读访问完成。
- call_rcu()：注册一个回调函数，当所有现存的读访问完成后，调用这个回调函数销毁旧数据。

RCU的一个例子

CPU0	CPU1
g_ptr=old_data	
rcu_read_lock()	新分配一个数据 修改新数据 g_ptr指向新数据
p=g_ptr=old_data	
	call_rcu()注册回调函数。 当所有读者线程都执行完毕， 回调函数销毁旧数据
rcu_read_unlock()	



读临界区指针副本p指向旧数据



g_ptr指向新数据，grace period结束后释放old指针指向的老数据

Documents/RCU/whatisRCU.txt

实验

实验1：自旋锁

- 实验目的
  - ✓ 了解和熟悉自旋锁的使用。
- 实验步骤
  - ✓ 在自旋锁里面，调用alloc_page(GFP_KERNEL)函数来分配内存，观察会发生什么情况。
  - ✓ 手工创造递归死锁，观察会发生什么情况。
  - ✓ 手工创造AB-BA死锁，观察会发生什么情况。

实验2：互斥锁

- 实验目的
  - ✓ 了解和熟悉互斥锁的使用。
- 实验步骤
  - ✓ 在第5章的虚拟FIFO设备中，我们并没有考虑多个进程同时访问设备驱动的情况，请使用互斥锁对虚拟FIFO设备驱动程序进行并发保护。
  - ✓ 我们首先要思考在这个虚拟FIFO设备驱动中有哪些资源是共享资源或者临界资源的。

实验3：RCU

- 实验目的
  - ✓ 了解和熟悉RCU锁的使用。
- 实验步骤
  - ✓ 编写一个简单的内核模块，创建一个读者内核线程和一个写者内核线程来模拟同步访问共享变量的情景。

BACKUP

第1季旗舰篇课程目录

课程名称	时长
序言一： Linux内核学习方法论	0:09:13
序言二： 学习前准备	
序言2.1 Linux发行版和开发板的选择	0:13:56
序言2.2 搭建Qemu+gdb单步调试内核	0:13:51
序言2.3 搭建Eclipse图形化调试内核	0:10:59
实战运维1：查看系统内存信息的工具（一）	0:20:19
实战运维2：查看系统内存信息的工具（二）	0:16:32
实战运维3：读懂内核log中的内存管理信息	0:25:35
实战运维4：读懂 proc meminfo	0:27:59
实战运维5：Linux运维能力进阶线路图	0:09:40
实战运维6：Linux内存管理参数调优(一)	0:19:46
实战运维7：Linux内存管理参数调优(二)	0:31:20
实战运维8：Linux内存管理参数调优(三)	0:22:58
运维高级如何单步调试RHEL— CENTOS7的内核一	0:15:45
运维高级如何单步调试RHEL— CENTOS7的内核二	0:41:28
vim:打造比source insight更强更好用的IDE(一)	0:24:58
vim:打造比source insight更强更好用的IDE(二)	0:20:28
vim:打造比source insight更强更好用的IDE(三)	0:23:25
实战git项目和社区patch管理	
2.0 Linux内存管理背景知识介绍	
奔跑2.0.0 内存管理硬件知识	0:15:25
奔跑2.0.1 内存管理总览一	0:23:27
奔跑2.0.2 内存管理总览二	0:07:35
奔跑2.0.3 内存管理常用术语	0:09:49
奔跑2.0.4 内存管理究竟管些什么东西	0:28:02
奔跑2.0.5 内存管理代码框架导读	0:38:09
2.1 Linux内存初始化	
奔跑2.1.0 DDR简介	0:06:47
奔跑2.1.1 物理内存三大数据结构	0:19:39
奔跑2.1.2 物理内存初始化	0:11:13
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31
奔跑2.1 代码导读C语言部分（一）	0:27:34
奔跑2.1 代码导读C语言部分（二）	0:21:28
2.2 页表的映射过程	
奔跑2.2.0 ARM32页表的映射	0:08:54
奔跑2.2.1 ARM64页表的映射	0:10:58
奔跑2.2.2 页表映射例子分析	0:11:59
奔跑2.2.3 ARM32页表映射那些奇葩的事	0:09:42
2.3 内存布局图	
奔跑2.3.1 内存布局一	0:10:35
奔跑2.3.2 内存布局二	0:13:30
2.4 分配物理页面	
奔跑2.4.1 伙伴系统原理	0:10:10
奔跑2.4.2 Linux内核中的伙伴系统和碎片化	0:11:14
奔跑2.4.3 Linux的页面分配器	0:21:37
2.5 slab分配器	
奔跑2.5.1 slab原理和核心数据结构	0:18:36
奔跑2.5.2 Linux内核中slab机制的实现	0:16:56
2.6 vmalloc分配	
奔跑2.6 vmalloc分配	0:15:48
2.7 VMA操作	
奔跑2.7 VMA操作	0:16:42
2.8 malloc分配器	
奔跑2.8.1 malloc的三个迷惑	0:17:41
奔跑2.8.2 内存管理的三个重要的函数	0:17:38
2.9 mmap分析	
奔跑2.9 mmap分析	0:23:14
2.10 缺页中断处理	
奔跑2.10.1 缺页中断一	0:31:07
奔跑2.10.2 缺页中断二	0:16:58
2.11 page数据结构	
奔跑2.11 page数据结构	0:29:41
2.12 反向映射机制	
奔跑2.12.1 反向映射机制的背景介绍	0:19:01
奔跑2.12.2 RMAP四部曲	0:07:31
奔跑2.12.3 手撕Linux2.6.11上的反向映射机制	0:07:35
奔跑2.12.4 手撕Linux4.x上的反向映射机制	0:10:08
2.13 回收页面	
奔跑2.13 页面回收一	0:16:07
奔跑2.13 页面回收二	0:11:41
2.14 匿名页面的生命周期	0:26:16
2.15 页面迁移	0:19:07
2.16 内存规整	0:24:03
2.17 KSM	0:28:17
2.20 Meltdown漏洞分析	
奔跑2.20.1 Meltdown背景知识	0:10:13
奔跑2.20.2 CPU体系结构之指令执行	0:11:25
奔跑2.20.3 CPU体系结构之乱序执行	0:11:03
奔跑2.20.4 CPU体系结构之异常处理	0:03:48
奔跑2.20.5 CPU体系结构之cache	0:10:56
奔跑2.20.6 进程地址空间和页表及TLB	0:17:39
奔跑2.20.7 Meltdown漏洞分析	0:06:04
奔跑2.20.8 Meltdown漏洞分析之x86篇	0:12:07
奔跑2.20.9 ARM64上的KPTI解决方案	0:25:39
代码导读	
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31
奔跑2.1 代码导读C语言部分（一）	0:27:34
奔跑2.1 代码导读C语言部分（二）	0:21:28
代码导读3页表映射	1:12:40
代码导读4分配物理页面	0:55:57
git入门和实战	
git入门与实战：节目总览	0:08:48
git入门与实战1：建立本地的git仓库	0:30:53
git入门与实战2：快速入门	0:12:45
git入门与实战3：分支管理	0:24:27
git入门与实战4：冲突解决	0:20:20
git入门和实战5：提交更改	0:12:15
git入门和实战6：远程版本库	0:13:26
git入门和实战7：内核开发和实战	0:15:52
git入门和实战8：实战rebase到最新Linux内核代码	0:18:07
git入门和实战9：给内核发补丁	0:13:57

第2季旗舰篇课程目录	
课程名称	时长
进程管理	
进程管理1基本概念	0:52:16
进程管理2进程创建	0:53:24
进程管理3进程调度	0:54:51
进程管理4多核调度	0:49:38
中断管理	
中断管理1基本概念	1:04:27
中断管理2中断处理part1	0:46:28
中断管理2中断处理part2	0:10:19
中断管理3下半部机制	0:55:57
中断管理4面试题目	1:13:57
锁机制	
锁机制入门1基本概念	0:56:16
锁机制入门2-Linux常用的锁	0:54:01

实战死机专题课程目录	
课程名称	时长
上集x86_64	
实战死机专题（上集）part1-kdump+crash介绍	0:30:09
实战死机专题（上集）part2-crash命令详解	0:28:15
实战死机专题（上集）part3-实战lab1	0:12:38
实战死机专题（上集）part4-实战lab2	0:11:03
实战死机专题（上集）part4-实战lab3	0:06:48
实战死机专题（上集）part4-实战lab4	0:15:28
实战死机专题（上集）part4-实战lab5	0:12:21
实战死机专题（上集）part4-实战lab6	0:24:07
实战死机专题（上集）part4-实战lab7	0:59:34
下集arm64	
实战死机专题(下集)part1	0:13:19
实战死机专题(下集)part2	0:20:47
实战死机专题(下集)part3	0:11:22
实战死机专题(下集)part4	0:33:01

全程约5小时高清，140多页ppt，8大实验，基于x86_64的**Centos 7.6**和**arm64**，提供全套实验素材和环境。全面介绍kdump+crash在死机黑屏方面的实战应用，全部案例源自线上云服务器和嵌入式产品开发实际案例！

微店 奔跑吧Linux内核

微店二维码

淘宝店二维码

《奔跑吧Linux内核 * 入门篇》相关的免费视频，或者更多更精彩更in的内容，请关注奔跑吧Linux社区微信公众号