《openEuler内核编程》

课程讲稿

第四章 第4讲

同步

软件所制

第四章 第4讲 同步

**学时：**2学时

**教学目的：**系统学习同步

**PPT讲稿：**



今天的内容是第四讲，同步。这也是操作系统构建机制里非常重要的功能。



这一节分六个部分，我们将主要了解一下同步机制、临界区、原子操作、内存屏障、锁的概念，信号量和条件变量。



首先一个问题，多线程之间是如何传递数据的？

那多进程没有共享地址空间的时候呢？他们怎么互相传递数据？

这就需要内核的参与。

内核为他们分配、映射共享内存、构建通信通道，从而保证数据的正确传递。

那同步，其实就是这中间保证信息传递过程中正确的方法

在操作系统中，共享资源可能包括变量、缓冲区、文件、甚至设备

那这些共享资源怎么进行访问控制、并发操作如何管理调度、指令执行的控制都是同步机制需要保障的

我们在来看一下并发冲突的定义

两个或两个以上执行主体对共享资源的冲突性竞争访问，导致执行结果不可预知，甚至对系统运行产生破坏性影响

产生并发冲突需要三个条件：资源共享、时间重叠、状态更改

操作系统存在大量并发冲突需要解决，而同步机制就是管理并发的基础设施

既然我们已经知道并发冲突产生的三个条件，那我们解决他的基本思路，

就是破坏资源共享条件，破坏时间重叠条件。

状态更改条件是执行主体的客观要求，一般不能更改

这个表里列出了一些常用的解决并发冲突的技术手段。所有其他新出现的手段基本上都是基于这些基本技术形成的。

那我们主要要介绍的。就是这两种解决机制的相关技术。包括一些典型策略问题。如生产者-消费者问题、读者-写者问题等。

第二部分内容是临界区和原子操作。

临界区的定义是两个或以上的进程不能同时访问的资源。

假设一个简单的飞机订票系统有两个终端T1和T2。

考虑进程T1和T2是并发执行的。当系统中只有最后一张票的情况下，即n=1时，T1卖出一张票。假设T1正在执行“卖一张票”的同时，进程T2正好执行条件判断，因为此时n=1的条件判断可以通过，故进程T2也可以卖一张票。这样一张票就卖给了两个顾客。



临界区：多个进程对于临界资源的访问必须互斥的进行。而每个进程中访问临界资源的那段代码就被称为临界区。

进入区：临界资源需要被互斥的访问，实际上是要求进程能够互斥的进入临界区。因此在临界区之前，需要对欲访问的临界资源进行检查，实现这段检查功能的代码被称为进入区。(资源已使用，则不能进入临界区。资源未被使用，则访问资源，并修改访问标志。)

这张图给出了经典临界区互斥的过程，



所谓互斥，就是：

多个进程不能同时使用一个临界资源

即两个或两个以上进程必须互斥的使用临界资源。

进程之间由于竞争同一资源而产生的相互制约关系，称为进程的互斥

所谓同步，就是：

有协作关系的进程之间不断地调整它们之间的相对速度和执行过程，以保证临界资源的合理利用和进程的顺利执行。

实现进程同步机制的方法可能有所不同，通常借助一个中间媒体来实现，如信号量、加锁操作等等。



同步机制的基本设计规则，包括空闲让进、和忙则等待。

19:

而对于等待本身，又分为有限等待和让权等待

20:

好，我们对于同步和互斥的基本概念有了了解，那我们现在来介绍原子操作。

原子操作是指一次不存在任何中断或失败的操作

对临界区的操作必须是原子操作

操作系统需要利用同步机制在并发执行的同时，保证一些操作为原子操作

21:

这是一个例子，和上面订飞机票的例子很像。

22:

系统硬件对“加1操作”的执行是非原子性的

系统中存在非原子的硬件指令

执行分为读、改、写等多个三个环节

执行时间窗口内可以与其它访问重叠交织

对策：CPU的原子指令

破除冲突的时间条件

原子性指令执行过程是排他性的

执行时间窗口与其它访问不重叠

23:

对应的，在操作系统中。

操作系统定义专用的原子操作，并充分利用CPU的原子性指令支持。

24:

这张图给出了一些基本的整数型原子操作，包括赋值、加、减等操作

25:

这张图给出了一些基本的位级原子操作。

26:

我们看其中的一个具体实现。

发现内核中的原子add操作，使用了LOCK\_PREFIX这个宏。来锁定特定内存，保证对特定内存的互斥访问。

其中，这个asm volatile是让编译器不要优化这一行代码。因为编译会造成乱序，从而导致原子操作不再是原子的。

27:

Ok 接下来我们讲解内存屏障。

28:

这是一个典型的乱序执行对并发造成的影响。结果不一致。

我们刚学了一种做法，是怎样？

对于需要指定操作的，要让他原子化，对吧？

那现在，我们用一些不一样的方法。

29:

乱序执行一般是因为编译器的指令调度优化或者现代处理器流水线的乱序执行指令

30:

当前的一些存储一致性模型，包括cache的和内存的一致性模型。

31:

那这些模型都是用到了一些屏障原语

分为控制编译乱序和控制流水线执行乱序两类。屏障前后执行顺序与源码一致

32:

这个是一个针对编译器的屏障原语。

volatile关键字通知编译器不能对本段代码乱序混排，保证了barrier前后的源码执行的有序

memory关键字通知编译器本处可能会更改所有内存 状态， barrier之前的存放于寄存器中的值都已经无效，因此barrier之后的所有变量的值都需要从内存中重新获取，杜绝了乱序执行的可能

33:

Mb这个宏，主要是为了保证指令执行顺序，不会因为多级流水线的乱序执行打乱。

34:

好。

我们学完了同步和互斥的基本概念，临界区、原子操作，以及屏障原语。

现在，我们来学习锁。

35:

设计思路：破除同步冲突的时间重叠条件、用锁实现在时间上的并发隔离、锁是一种软件机制，实现相对复杂，需要硬件指令支持

使用场景：适合于控制执行时间较长的临界区

相较于原子操作：原子操作功能单一，锁功能全面；原子操作开销小，锁的开销较大

典型锁包括自旋锁、读写锁、信号量、RCU

36:

锁是一个内存中的对象，提供两种基本操作

Acquire(): 和Release():

进程pair calls来获取和释放锁

在acquire/release之间，线程hold锁。直到锁的拥有者释放锁，acquire才可以返回

如果不是pair calls？

会造成一直hold住锁不释放。

37:

当蓝色的进程acquire锁会发生什么？

为什么返回在临界区外面？

如果第三个线程acquire锁，会发生么

38:

这是一个典型的too much milk例子

39:

通过便签note约定使用冰箱

进程在检查便签和note之后，被切换会发生什么？

40:

一种解决方案，两个人采取不同的处理流程。

解决了，但是太复杂，且浪费资源。

41:

非原子操作，会被其他程序打断，如中断，上下文切换

临界区太大：冰箱中还有其他食品时，别人无法取到

42:

我们进一步来解决。

将检查note和买牛奶的操作分离，只对检查操作进行锁

43:

怎么实现锁？下面是一个例子

这个锁的实现方式叫spinlock，

44:

自旋锁的设计原理：若锁未被占用，则立刻获得锁，并继续执行；若锁被占用，忙等，等待锁被释放；阻止两个或以上的进程同时进入临界区

自旋锁更加轻量级、只有单一拥有者，只能加锁一小段时间

可以在中断上下文中使用，但信号量不可以，why？

因为信号量会造成睡眠。这个我们在后面会继续介绍。

45:

这是Linux中自旋锁的实现。

我们看到有不同情况下的实现。主要有对中断禁用的一个变体。

46:

两个进程会同时注意到一个锁被释放了

那怎么解决呢？

让锁操作具有原子性

47:

回顾我们之前对于临界区的介绍。

在操作系统执行过程中，为了保障临界区，包括主要三种方式。

禁用中断，利用硬件的原子操作指令、以及基于软件方法的实现。

48:

最简单粗暴的就是禁用中断来实现互斥。

49:

我们看这个例子。

那他的问题是什么呢？

没有了中断，内核无法轮转调度，其他线程会处于饥饿状态，永久等待

50:

使用锁变量，并只在对锁变量进行测试和赋值时通过中断实现互斥

是否解决了永远等待的问题？？

51:

我们继续引入队列的方式。

加入了等待队列

主动yield，放弃cpu执行资源， 这属于我们前面讲的维护同步机制中的“让权等待”

52:

禁止中断，只能由内核执行，为什么？

首先是权限的问题。只有在ring 0情况下，才可以执行中断禁止的操作。

其次是安全的考量，如果用户态应用随时禁止中断，会造成其他进程饥饿

禁止中断的方法对于多核处理器适用吗？

禁用中断后，进程无法被停止 – 整个系统都会为此停下来  
– 可能导致其他进程处于饥饿状态

• 临界区可能很长  
– 无法确定响应中断所需的时间(可能存在硬件影响)

多核的时候，当一个CPU将中断屏蔽后,只影响当前屏蔽中断的CPU,其他CPU还是依然可以照样访问内存的

唯一一个当一个CPU在访问内存时阻止其他CPU访问内存的方法就是将内存总线锁住,这个需要硬件的支持,TSL可以达到该目的。

53:

好，第二种方式，原子操作指令实现锁，来保障临界区

就是测试和置位(Test-and-Set，TAS/TS)指令

执行过程中，如果flag的初始值是false的话，执行结束后，其值为什么？真？

返回值呢？如果初始值是false，执行结束后，返回什么？

54:

硬件提供了锁内存总线的机制,我们在锁内存总线的状态下执行test and set操作,就能保证同时只有一个核来test and set.

55:

如何使用交换指令来实现锁？？

56:

用TAS指令实现锁

57:

无忙等锁，相当于让权等待锁。

58:

基于软件方法的忙等待。

两个线程，T0和T1。通过共享一些共有变量来同步

59:

Peterson算法，最早是1981年提出的，满足线程间互斥的软件解决方案。

60:

这是dekkers算法。

61:

多线程的软件方法。

很多现代软件架构都借鉴了这种机制，如xenloop等

问题： 比较负载，需要两个进程/线程之间共享数据项；需要忙等待，浪费CPU时间

62:

好

我们总结一下自旋锁的优缺点

优点包括

适用于单处理器或者共享主存的多处理器中任意数量 的进程同步

简单并且容易证明

支持多临界区

缺点包括

忙等待消耗处理器时间

可能导致饥饿和死锁

63:

读写自旋锁，是一种效率更高的变种锁，分为读锁、写锁

读锁允许多个读进程同时拥有锁

写锁只允许1个拥有者，且不允许并发读

64:

这是Linux中的实现，我们可以和自旋锁的实现进行对比

65:

读写锁的设计实现。

由32位数据表示，含两部分编码

第0-23位：读者数量

第24位：可写标记，为1表示为可写

66:

读写锁的问题？

缺乏公平性，读者容易饿死写者！

67:

Seqlock

一种“写操作”优先的锁，解决普通read\_lock读写优先级相同的问题

实现机制：锁带序列号，每次写时加1；读数据前后判断序列号是否变化，有变化则重试；并发写会使读操作循环重复，直到写锁已释放

使用范围

大量读、少量写；优先写操作，避免读操作阻塞写操作

典型场景：jiffies变量的更新

68:

Seqlock的使用例子

读者在读取数据前后，需要两次加锁获取锁

对前后获取锁的seq进行比较，值相等判断加锁成功

69:

锁的弊端：读写锁比常规锁更快，但获取读锁仍然开销昂贵；导致忙等、阻塞；持有锁时间长时，所有的读者仍然需要等待；当涉及到对时间敏感的操作时，不好采用 ；可扩展性差

改进思路：lock free

破除并发冲突的资源共享条件

完全消除锁

70:

RCU锁得设计思路：数据的读版本和写版本分离

优势：没有锁，也没有死锁

读者不会阻塞；写者更新数据时也不会阻塞；开销：需要垃圾回收（GC）

71:

这张图给出了RCU锁链表删除的过程

72:

RCU锁链表更改的操作。如图所示

进程p访问链表元素；进程q需要更新链表元素，分配一个链表元素的存储空间；进程q拷贝一份副本

73:

进程q将副本内容进行更新；更新链表指针，将跟新后的元素链入链表；删除旧的元素

74:

这是RCU锁的函数接口。

包括了：

读者相关和写者相关的接口，以及回收相关的接口。

75:

RCU的垃圾回收：需要确认对数据的reference都已完成，即“访问清空”；synchronize\_rcu“等待清空”；call\_rcu“回调机制”实现数据回收

非抢占内核下：实现非常简单，相当多函数为空，开销低：进程切换保证进入了quiescence状态

抢占内核下

采用引用计数方式(reference counting)；实现复杂，通过软中断tasklet等机制完成清空与回调

76:

这是非枪战内核下的RCU实现。

77:

优势：读者永远不阻塞,写者永远不阻塞；CPU核数极多时，仍然具有良好的扩展性；没有死锁

弊端

适合类似指针链表场景有限；当存在多个并发写者时仍然需要额外机制进行同步 ；需要维护多个数据版本 ；一般用高层次API进行编程

例如 list API、Tree API

在Linux内核中的使用情况：2002年35处使用；2012年超过10000处