《openEuler内核编程》

课程讲稿

第四章 第5讲

信号量

软件所制

第四章 第5讲 信号量

**学时：**2学时

**教学目的：**系统学习信号量

**PPT讲稿：**



今天的内容是第5讲，信号量。



这一节分4个部分，我们将主要介绍生产者和消费者问题，信号量、条件变量的概念和使用方式，以及一些常见的同步问题。



生产者：生产资源，并将资源供给消费者进程使用

消费者：使用消耗资源

缓冲区：固定大小，用以缓存生产者生产的资源

生产者和消费者执行在不同的频率：没有执行顺序要求；任务之间相互独立；缓冲区不需要明确的交接手续

同步：怎样保证生产者和消费者以正确的方式访问共享缓冲区？

在内核中广泛使用。E.g. I/O

为什么不用if，而一定要用while

如果使用if()。对于简单逻辑没问题。

我们考虑单一生产者，多个消费者的情况。如果消费者a被唤醒后执行，但在他运行前，消费者b已经改变了缓冲区的状态。

所以使用while就安全了。

利用锁实现对缓冲区的互斥读写访问

这样的实现有什么问题吗？

临界区太大？ 拆分到只锁检查？

Busy waiting？ 浪费资源

我们引入新的同步原语

忙等待带来大量资源浪费

将操作和检查分离，只锁检查操作？

需要更强力的同步原语——只需要做简单的检查和等待（睡眠）

High-level 同步机制：信号量 和条件变量

我们来介绍一下信号量

信号量是带睡眠功能的同步机制

当进程尝试获取一个被占用的信号量时，进程会被放入等待队列，进程进入睡眠状态；用于实现对进程执行顺序的控制，如生产者进程-消费者进程

最早由Edsger Dijikstra在上世纪60年代中发明

信号量是一个非负整型的同步变量

他有两个主要的原语操作：

Wait和signal，即pv操作

Pv操作是因为荷兰语的读音

P代指荷兰语中的to probe

V代指荷兰语中的to test。

有时候人们也会称其为下（down）和上（up）操作

每个信号量都关联了一个等待进程队列

当一个线程调用wait():如果信号量open，线程继续执行；如果信号量closed，线程阻塞到队列，睡眠

Signal()原语open信号量：：如果一个线程在等待队列中，线程被唤醒；如果没有线程在等待队列，signal会被记录等待下一个线程

信号量的数据结构分类包括整型信号量、记录型信号量、AND型信号量和信号量集



Binary信号量：只取0或1

是否与锁相似？

S=0 有人hold拥有了这个锁

三个进程同时进行信号量操作时，因为S=1，最多只有一个可以hold住信号量，进行p操作



当S=2的时候，可以多个进程同时进行p操作，



整型信号量的数值：表示当前系统可用该类型临界资源的数量

如果设置整型信号量s, s值得含义包含以下几种：

S>0 表示系统中空闲的该类临界资源个数。

S=0 表示系统中该类临界资源刚好被全部占用，同时没有进程在等待临界资源。

S<0 S的绝对值表示系统中等待该类临界资源的进程个数

19:

记录型信号量中S的值

是该类临界资源的数量，L为进程链表指针，指向等待该类资源的PCB队列。

20:

AND信号量集：同时需要多个资源且每种占用一个资源时的信号量操作

将进程运行过程中所需要的所有资源，一次性全部分配给进程

待进程使用完成以后再一起释放。

只要有一个资源尚未分配给进程，则其他可能分配的资源也不能分配给它。

21:

一般“信号量集”是指同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的信号量处理。由于—次需要n个某类临界资源，因此如果通过n次wait操作申请这n个临界资源，操作效率很低，并可能出现死锁。—般信号量集的基本思路就是在AND型信号量集的基础上进行扩充，在一次原语操作中完成所有的资源申请。进程对信号量Si的测试值为ti(表示信号量的判断条件，要求Si>ti即当资源数量低于ti时，便不予分配)，占用值为di（表示资源的申请量，即Si=Si—di)。对应的P、V原语格式为：

22:

信号量的使用：适用于被占用较长时间的锁；短时间的加锁场景不适合采用信号量；维护等待队列、切换上下文会有开销

信号量只能在进程上下文中使用，不可用于中断上下文，why？ 因为会造成睡眠

在获取信号量时，不能拥有自旋锁：获取信号量不成功时，会出现“带锁睡眠”

保证同时只有一个进程可以访问共享数据，使用哪种信号量？

允许进程等待特定条件发生，使用哪种？

23:

相较于锁，信号量有更多的语义

当信号量大于1，可以允许多个进程同时访问临界资源

当信号量等于1，可以用来做互斥访问

24:

我们发现linux中对信号量的实现，是基于spinlock的！

25:

其中的pv操作，包括up、down和可中断和非阻塞的down操作。

26:

Up操作的实现

释放信号量，count++，并唤醒睡眠的进程

27:

Down操作的实现

Count--

28:

加入阻塞等待队列，设置任务不可中断，执行调度，让出CPU执行资源

29:

类似读写锁，我们也有读写信号量。

内核按FIFO的顺序处理信号量的申请

当信号量被写进程占用时 ，所有的其它等待进程进入睡眠

当信号量被读进程占用 ，在第一个写进程执行前的所有读进程可以获得信号量

30:

Mutex是Linux中一种精简高效的信号量实现；普通信号量过于通用化和复杂，适用于共享资源数多的情况；一般情况下，采用mutex，除非有特殊限制

尝试回忆，mutex其实就是一种binary信号量！

31:

这是Mutex的函数接口

mutex\_lock和mutex\_unlock总是成对出现，是Linux内核中使用最广的同步机制之一

32:

Mutexcount的三种状态

1:表示空闲

0:表示上锁

负数：表示上锁，且有等待者

33:

Mutex包括三种执行路径

Fastpath、midpath和slowpath

由于midpath避免了等待者立刻睡眠，在实际场景下 mutex的性能要显著优化信号量

34:

这是一个mutex和信号量的性能对比，

几乎是信号量的两倍还多。

35:

我们对Linux信号量做一次总结分析

36:

除了刚才所说的信号量外，linux还实现了完成变量。

一种更简洁的信号量

适用场景：两个进程间的通信同步

进程A等待另一个进程B完成某项工作

进程B完成后，唤醒进程A

37:

这张表介绍了各种锁和信号量的使用建议。

38:

而除了内核中的信号量外，Linux也给用户态应用提供了同步机制futex。一种用户态的信号量

一部分数据结构设置在用户空间，可供进程或线程在用户态直接 访问

必要的数据结构如进程队列放置于内核空间

39:

Linux下的futex系统调用非常常用，pthread库下的许多同步机制都是通过futex完成的

核心机制：普通的加锁操作直接在用户态完成；只有当需要睡眠、唤醒时才进入内核

Futex自身是一个int型变量，由addr1指针指向。在所有平台上都是32位

Futex的值完全由应用程序控制，Futex可以使用普通的内存地址，唯一要求是按int对齐

40:

Futex的三种典型操作

Wait、wake和cmp\_requeue

41:

好，介绍完了信号量，我们再来介绍一下条件变量

42:

回顾一下生产者-消费者问题

我们使用两个独立的信号量来限制

Full = 0

Empty = N

为什么使用两个？

原因是有多个缓冲区可供消费，而不是有多个消费者。

以后引入的mutex是用来保证循环缓冲区的管理。

信号量和锁保护的共享资源，与进程数量（只要大于1）无关。

因为消费者以Empty信号量为准，empty = N， 大于1，多个线程可以同时操作缓冲区资源，无法对临界资源进行保护！！！！！

43:

我们增加mutex保护临界区

存在什么问题？？

虽然保证了临界区的安全，一次只有一个进程可以操作缓冲区资源

但是当一个线程带锁睡眠的时候，会发生死锁！！！！

44:

我们重新定义一下临界区域

45:

条件变量这一概念的引入。

是针对环形缓冲区的生产者和消费者问题

空或者满的状态判断通过锁机制进行保护 ；缓冲区的读写操作通过锁机制进行保护 ；消费者/生产者的睡眠和唤醒通过信号量实现

单一使用锁或者信号量无法实现上述要求

单靠锁过于极端，无效等待过久

靠“信号量+锁”，加锁使用信号量又面临带锁睡眠问题造成死锁

46:

当队列已经为空时，生产者进程长时间不生产

消费者进程不知道需要等待多长时间，通过无效循环很久等待队列非 空，极大耗费CPU资源

47:

48:

条件变量是基于锁的互斥机制和基于条件变量的睡眠唤醒机制

类似于“锁+信号量”，睡眠前自动释放锁，唤醒后重新加锁

与信号量的区别：在拥有锁的临界区内；不能使用信号量，因为会可能睡眠而死锁；但可以使用条件变量，解决临界区内“带锁睡眠”问题

49:

Broadcast开销较大，但是比指定的好处是没有那么复杂的逻辑

50:

这是pthread库中的条件变量操作

51:

通过环形缓冲区进行数据共享

52:

加锁，实现对共享缓冲区的保护

判断缓冲区是否为满

如果满，则生产者在条件变量b->more上睡眠。

在睡眠时，wait条件变量会自动释放mutex，被唤醒时，会重新获取mutex。

53:

若缓冲区为空，则消费者在条件变量上睡眠。

在睡眠时，条件变量自动释放mutex，被唤醒时，重新获取mutex

54:

好，我们接下来看几个常见的同步问题

55:

读者-写者问题：

文件F可以被多个并发进程共享。按访问方式，可以分为两类。

只能读共享对象内容的进程，称为读者进程。

要更新共享对象文件的进程，称为写者进程

56:

读者reader只能在没有写者writers的情况下执行

Writers只有在没有其他读者/写者的情况下执行

wmutex。互斥使用该共享文件信号量。共享文件只有一个，初始情况下，值为1。

readcount整型变量。表示正在读的信号量，初始值为0

rmutex计数器readcount的信号量，因为readcount是一个可被多进程访问的临界资源。初始状态无进程读写，值为1。

57:

读者进程执行顺序

58:

写着进程通过条件变量和mutex实现

59:

哲学家进餐问题

假设5个哲学家坐在一张圆桌前吃饭，桌上5支筷子，每人之间各一支。吃饭时，必须分别从左、右两边拿到筷子，才能吃饭。如果筷子已在他人手上，则该哲学家必须等待他人吃完才能拿筷子，任何一个哲学家自己未拿到两只筷子吃饭前，绝不放下自己手里的筷子。描述5位哲学家吃饭的进程

60:

筷子是临界资源，一段时间只允许一位哲学家使用，

为了实现对筷子的互斥使用，可以为每一支筷子设计一个信号量

Var chopstick: array[0,…,4] of semaphore

初始条件下，所有哲学家都未吃饭，估信号量为1。

61:

这个方法有问题吗？

62:

当5个哲学家同时拿起左边的筷子时，就没有任何一个哲学家能够拿到右边的筷子。这种情况就形成了死锁。

63:

有没有什么方法可以避免死锁呢？至多允许4位哲学家同时拿左边的筷子，这样能保证至少有一位哲学家进餐。仅当哲学家左、右两支筷子均可用时，才允许他拿起筷子。规定奇数号哲学家先拿左边的筷子，再拿右边的，而偶数号则相反，这样总有一位哲学家能够获得两只筷子。

64:

完