同步

背景

多个进程并发访问和操作同一数据并且执行结果与特定访问顺序有关， 称为竞争条件。
为了防止竞争条件， 需要确保一次只有一个进程可以操作变量 counter 为了做出这种保证， 要求这些进程按一定方式来同步

协作进程如何进行进程同步 和进程协调

临界区问题

每个进程有一段代码， 称为临界区 进程在执行该区时可能修改公共变量、 更新一个表、 写一个文件等。 该系统的重要特征是， 当一个进程在临界区内执行肘， 其他进程不允许在它们的临界区内执行。

临界区问题 是， 设计一个协议以便协作进程。 在进入临界区前， 每个进程应请求许可。 实现这一请求的代码区段称为进入区 临界区之后可以有退出区，其他代码为剩余区

临界区问题的解决方案应满足如下三条要求：
- 互斥 : 如果进程户,在其临界区内执行， 那么其他进程都不能在其临界区内执行。
- 进步 : 如果没有进程在其临界区内执行， 并且有进程需要进入临界区， 那么只有那些 不在剩余区内执行的进程可以参加选择， 以便确定谁能下次进入临界区，而且这种选择不能无限推迟。
- 有限等待: 从一个进程做出进人临界区的请求直到这个请求允许为止， 其他进程允许进人其临界区的次数具有上限。

两种常用方法， 用于处理操作系统的临界区问题： 抢占式内核 与非抢占式内核

Peterson 解决方案

变量 turn 表示哪个进程可以进人临界区。 即如果 turn==i, 那么进程pi允许在临界区内,数组 flag 表示哪个进程准备进人临界区。 例如， 如果 flag[i]为 true, 那么进程pi， 准备进人临界区。

硬件同步

临界区问题的多个解答， 这包括内核开发人员和应用程序员采用的硬件和软件 API 技术。所有这些解答都是基于加锁 为前提的， 即通过锁来保护临界区。

在多处理器环境下， 这种解决方案是不可行的。 多处理器的中断禁止会很耗时，因为消息要传递到所有处理器。 消息传递会延迟进人临界区， 并降低系统效率。另外， 如果系统时钟是通过中断来更新的， 那么它也会受到影响。

许多现代系统提供特殊硬件指令， 用于检测和修改字的内容， 或者用于原子地 交换两个字（作为不可中断的指令)。 我们可以采用这些特殊指令， 相对简单地解决临界区问题。

互斥锁

采用互斥锁保护临界区,从而防止竞争条件,也就是说，一个进程在进人临界区时应得到锁，它在退出临界区时释放锁，函数acquire（）获得锁，函数release释放锁

每个互斥锁有一个布尔变量 available, 它的值表示锁是否可用。 如果锁是可用的，那么调用acquireO 会成功， 并且锁不再可用。 当一个进程试图获取不可用的锁时， 它会阻塞， 直到锁被释放。

忙等待。 当有一个进程在临界区中， 任何其他进程在进人临界区时必须连续循环地调用 acquire()。 其实， 这种类型的互斥锁也被称为自旋锁， 因为进程不停地旋转， 以等待锁变得可用。

忙等待浪费 CPU周期， 而这原本可以有效用于其他进程。

自旋锁确实有一个优点： 当进程在等待锁时， 没有上下文切换（ 上下文切换可能需要相当长的时间）因此， 当使用锁的时间较短时，自旋锁还是有用的

信号量

一个信号置 S 是个整型变量， 它除了初始化外只能通过两个标准原子操作： wait()和 signal()来访问。

信号量的使用

操作系统通常区分计数信号量与二进制信号量。 计数信号量 的值不受限制， 而二进制信号量 的值只能为 0 或 1。 因此， 二进制信号量类似于互斥锁。 事实上，在没有提供互斥锁的系统上， 可以使用二进制信号量来提供互斥。

信号量的初值为可用资源数量。 当进程需要使用资源时， 需要对该信号量执行wait。操作（ 减少信号量的计数 ）。 当进程释放资源时， 需要对该信号量执行signal()操作（ 增加信号量的计数)。 当信号量的计数为 0 时， 所有资源都在使用中。 之后， 需要使用资源的进程将会阻塞， 直到计数大于 0。

也可以解决同步问题：synch 初始化为 0, 只有在p1 调用 signal(synch)， 即 s1 语句执行之后， s2才会执行

信号量的实现

当一个进程执行操作 wait。 并且发现信号量值不为正时， 它必须等待。 然而， 该进程不是忙等待而是阻塞自己。 阻塞操作将一个进程放到与信号量相关的等待队列中， 并且将该进程状态切换成等待状态。 然后， 控制转到 CPU 调度程序， 以便选择执行另一个进程

等待信号量 S 而阻塞的进程， 在其他进程执行操作 signalO 后， 应被重新执行。进程的重新执行是通过操作 wakeupO 来进行的， 它将进程从等待状态改为就绪状态。 然而， 进程被添加到就绪队列。

关键的是， 信号量操作应原子执行。 我们应保证： 对同一信号量， 没有两个进程可以同时执行操作 wait() 和 signal0。 这是一个临界区问题。 对于单处理器环境， 在执行操作wait() 和 signal()时， 可以简单禁止中断。

具有等待队列的信号量实现可能导致这样的情况： 两个或多个进程无限等待一个事件,而该事件只能由这些等待进程之一来产生。 这里的事件是执行操作 signalO。当出现这样的状态时， 这些进程就为死锁

死锁与饥饿

与死锁相关的另一个问题是无限阻塞或饥饿, 即进程无限等待信号量。 如果对与信号量有关的链表按 LIFO 顺序来增加和删除进程， 那么可能发生无限阻塞。

优先级的反转

优先级反转。 它只出现在具有两个以上优先级的系统中， 因此一个解决方案是只有两个优先级。

然而， 这对于大多数通用操作系统是不够的。 通常. 这些系统在解决问题时采用优先级继承协议。 根据这个协议， 所有正在访问资源的进程获得需要访问它的更高优先级进程的优先级， 直到它们用完了有关资源为止。 当它们用完时， 它们的优先级恢复到原始值。

经典同步问题

有界缓冲问题

读者 - 作者问题

假设一个数据库为多个并发进程所共享。 有的进程可能只需要读数据库， 而其他进程可能需要更新（ 即读和写） 数据库。 为了区分这两种类型的进程， 我们称前者为读者,称后者为作者。

如果两个读者同时访问共享数据， 那么不会产生什么不利结果。然而， 如果一个作者和其他线程（或读者或作者） 同时访问数据库， 那么混乱可能随之而来。

要求作者在写人数据库时具有共享数据库独占的访问权。 这一同步问题称为读者 - 作者问题

有些系统将读者- 作者问题及其解答进行了抽象， 从而提供读写锁

在获取读写锁时， 需要指定锁的模式： 读访问或写访问。 当一个程只希望读共享数据时， 可申读模式的读写锁； 当一 程希望修改共享数据时， 应申请写模式的读写锁。 多个进程可允许并发获取读模式的读写锁， 但是只有一个进程可获取写模式的读写锁 作者进程需要互斥的访问。

读写锁在以下情况下最为有用：
- 容易识别哪些进程只读共享数据和哪些进程只写共享数据的应用程序。
- 读者进程数比作者进程数多的应用程序。 这是因为读写锁的建立开销通常大于信号量或互斥锁的， 但是这一开销可以通过允许多个读者的并发程度的增加来加以弥补。

哲学家就餐问题

在多个进程之间分配多个资源， 而且不会出现死锁和饥饿

死锁问题有多种可能的补救措施：
- 允许最多 4 个哲学家同时坐在桌子上。
- 只有一个哲学家的两根筷子都可用时， 他才能拿起它们（ 他必须在临界区内拿起两根筷子)。
- 使用非对称解决方案。 即单号的哲学家先拿起左边的筷子， 接着右边的筷子；而双号的哲学家先拿起右边的筷子， 接着左边的筷子

管程

使用方法

只有管程内定义的函数才能访问管程内的局部声明的变量和形式参数。 类似地，管程的局部变量只能为局部函数所访问。

管程结构确保每次只有一个进程在管程内处于活动状态。 因此， 程序员不需要明确编写同步约束

对于条件变量， 只有操作 waitO 和 signalO 可以调用

操作 x.SignalO 重新恢复正好一个挂起进程。 如果没有挂起进程， 那么操作SignalO就没有作用， 即 x 的状态如同没有执行任何操作

假设当操作 x.signal() 被一个进程 P 调用时， 在条件变量 x 上有一个挂起进程Q。

唤醒并等待: 进程 P 等待直到Q 离开管程， 或者等待另一个条件

唤醒并继续: 进程Q等待直到离开管程或者等待另一个条件

哲学家就餐问题的管程解决方案

采用信号量的管程实现

管程内的进程重启

如果多个进程已挂起在条件 X 上，并且有个进程执行了操作 X.signal()， 那么我们如何选择哪个挂起进程应能重新运行?

条件等待（x.wait(c)）c为优先值

可能出现的问题
- 进程可能在没有首先获得资源访问权限时， 访问资源。• 进程可能在获得资源访问权限之后， 不再释放资源。
- 进程可能在没有请求之前， 试图释放资源。
- 进程可能请求同一资源两次（中间没有释放资源

确保正确的方法

为了确保系统正确， 我们必须检查两个条件。 第一， 用户进程必须总是按正确顺序来调用管程。 第二， 我们必须确保： 不合作的进程不能简单忽略管程提供的互斥关口， 在不遵守协议的情况下不能试图直接访问共享资源。

替代方法

事务内存

内存事务 为一个内存读写操作的序列， 它是原子的。 如果事务中的所有操作都完成了， 内存事务就被提交。 否则， 应中止操作并回滚。

优点
- 采用这种机制而不是锁的优点是： 事务性内存系统而非开发人员负责保证原子性。再者， 因为不涉及锁， 所以死锁是不可能的。
- 此外， 事务内存系统可以识别哪些原子块内的语句能并发执行， 如共享变量的并发读访问。

事务内存可以通过软件或硬件实现。

OpenMP

函数式编程语言

著名的编程语言， 如 C、 C++、 Java 和 C#, 称为命令式 (或过程式 语言。 命令式语言用于实现基于状态的算法。

命令式与函数式语言的根本区别是， 函数式语言并不维护状态。 也就是说， 一旦一个变量被定义和赋了一个值， 它的值是不可变的， 即它不能被修改。

优点 由于函数式语言不允许可变状态. 它们不需要关心诸如竞争务件和死锁等问题。