链滴

并发原理篇：锁

作者：tianxin

原文链接：https://ld246.com/article/1477323299224

来源网站：链滴

并发原理篇：锁

前言

最近在上操作系统的课，刚讲完了进程篇，但老师只是给我们勾画出进程的轮廓，所以在此总结进程发的相关内容，希望勾勒出她貌美的容貌。

这次内容主要分成三个模块去讲述 （也可以看目录哈）

- 并发面临的问题
- 如何解决问题——锁
- 锁的实现原理

何为进程，何为线程

在学习OS的时候，提到最多的一个概念便是进程，很多书中一上来就开始讲进程的三个状态图： 就，运行，阻塞。然而一开始就一脸蒙蔽。

关于进程，当时学习的时候有个人的小困惑：

- 进程在OS中的表现形式（存在形式）是什么?
- 进程到底是如何切换的?
- 进程状态管理在OS中又如何实现?

而这些和OS实现密切相关的问题在书上很少会找到答案。遗憾的是这篇文章并没有解释上述关于进的问题，（打算写成专门的一篇文章去介绍进程的实现与切换）。

如果从OS源码角度去探讨以上问题，那么OS的学习顺序则会转为： OS启动篇 —— OS 内存管理篇 —— OS 进程管理篇。

不过OS启动会涉及到太多太多硬件相关的内容，倘若并不是立志成为内核开发者或者是嵌入式开发，可以忽略OS启动代码，稍微了解启动过程即可。

作为一名学生党而言，十分推荐 清华大学的一门OS视频： 学堂在线——操作系统，同时讲师也是作系统——精髓与设计原理的译者。

并发面临的问题

先看一个简单的程序

```
int i = 0;
int increase() {
    i = i + 1;
    return i;
}
```

我们平时自己写程序而言，是没有任何问题的。但有多个线程去调用 increase方法，则会产生意外的情。

线程A在中途执行increase函数时会随机切换到B线程，这时线程A得到意外的结果 2， 而不是 1。

线程A和线程B可以说是形成了竞争 ： 对资源变量 i 使用的竞争，最终结果依赖于指令的执行顺序，生了竞争条件。

竞争条件 ： 如果运行的结果依赖于不同线程执行的先后的话，那么就会造成竞争条件(race condition)

我们该如何避免因竞争条件而造成的意外呢?

互斥与同步

注意互斥与同步而两个不同的概念。

上述问题分析

得到意外结果是原因是在A线程执行过程中被中断，同时B线程改变了二者共享的数据。

一种思路是说：避免共享数据的使用。（但有时候必须依靠全局变量）另一种思路：保证A线程在调用ncrease函数时不可以被打断去执行其他线程。

我们举一个生活中的例子：我们天天都会使用浴室，把浴室看做大家一起竞争的资源，我们如何避免己淋浴时他人进入浴室 —— 我们都会随手锁住门。

在OS中也是同样的道理，变量i是共享资源（也可是文件，打印机等其他设备），我们把对共享资源问的代码称为：临界区。为了避免多线程同时享受浴室，我们在访问资源的代码前加上一把锁： **表此时此刻，它是属于我滴，你们谁也进不来。**等到我洗浴完毕（执行完这段代码），才会打开这把锁。

这把锁叫做：**互斥锁。**被锁住的内容称为：**临界区。**同一时刻，只能有一个进程（线程）在临界区中行。所以通过互斥锁，我们实现了线程之间的互斥。

转化为伪代码是酱紫滴：打磨一把锁 -上锁 - 洗澡 - 开锁。

```
int i = 0;  // 共享浴室
lock l;     // 浴室里的那把锁
int increase() {
    lock(l);   // 上锁
    i = i + 1;
    unlock(l); // 解锁
}
```

通过使用互斥锁，只是实现了线程之间的互斥：无法同时访问临界资源。然而，如果我们想得到是 A远比B先执行（执行完A再执行B），那么互斥锁便无法满足这样的需求了 —— 如何达到线程之间的步呢？ 稍后解答!

锁的实现与分析

我们从代码的角度看问题：互斥锁是如何实现的呢?

方法1：基于硬件中断禁用

对于OS而言，进程的切换，用户程序执行过程产生的异常，读取IO结束后的反馈、键盘发送数据，一切的一切都是基于中断来实现的。

上述事件的发生，一律由我们（编写进程管理，负责异常处理，负责IO处理、键盘模块的工程师们）动调用一条汇编级别的中断指令 INT 8080， CPU会自动放下一切工作（放弃故乡，放弃学业，放弃

情，义无反顾的来到蓝翔，挖掘机哪家强，就来新东方）去调用位置固定的一段程序（中断入口程序，根据中断号 8080 找到对应的中断处理程序 （处理进程切换，处理异常等等）。

为了避免进程的切换，那么最直接了当的一个做法便是： 禁掉中断。

然而中断作为用户态和内核态交互的接口，断掉了它，便等于在这期间用户态和内核态之间是无法交的。所以只有最最简单的一些嵌入式领域可能会使用这样的方法。

总结： 禁用中断，就是断掉你进程切换的念头。但此法不是太可取：就像若想成功，必先子宫， 但实不必子宫，也可成功，那么接下来介绍不必子宫的秘籍。

方法2： 基于硬件指令

思路

对于每把互斥锁，设计一个变量表示锁的状态： 0 - 未锁定；1 - 已锁定。如果是上锁操作，则把 0 → 1，反之 1 → 0。

在上锁前，我们先检测锁的状态是不是还没有被锁住，如果被锁住了，则需要等到该锁的状态从 1 → 0 后，我们才能进行上锁。

转化为伪代码：

```
int status = 0;

// 尝试改变锁的状态 : 成功 or 失败; i 代表当前锁的状态。
boolean changeStatus(int i) {
    if (i == 0) { // 为锁定状态
        status = 1;
        return true;
    } else {     // 已锁定
        return false;
    }
}
```

上锁的操作：如果改变锁状态失败，那就不断重复，直到成功

```
void lock(...) {
    while (!changeStatus(status)) ;  // 如果状态失败，无限重复
    /* 临界区 */
    status = 0;           // 临界区执行完毕，将锁的状态重置为 0
```

这里对锁状态的改变，即 changeStatus函数 其实是一条的硬件指令（指令级别，不可中断）。（如状态改变的功能可以被打断，那还互斥什么）。 这条硬件指令称为： testset指令

总结： 本来是一系列的指令，由于可能被中途打断所以从硬件层面直接将几条指令设计为一条指令。

方法3： 基于软件方法

以下算法或许并不是作为互斥实现的主流算法，但适当的了解有助于我们在遇到其他并发问题时拓宽路。

思路一

我们可以使用一个全局变量 turn：表示 当前可以进入临界区的线程id 。

我们有两个线程： i 和 j。那如何基于软件方法实现两个线程的互斥呢?

伪代码思路：

```
/* 全局变量 */
int turn = i; // 初始化
```

● 对于线程i而言 ：

```
/* 线程 i */
while(turn != i);
/* 临界区 */
do something
turn = j;
```

● 对于线程j而言：

```
/* 线程 j */
while(turn != j);
/* 临界区 */
do something
turn = i;
```

这样通过变量 turn 的改变实现线程间的互斥。然而这样做有两个特别大的缺点：

● 当 turn != 线程id时，程序会不断的循环，这种过程称为**忙等待**或者**自旋等待**，最大的弊端是消CPU的资源（典型的占着xx不xx）

● 第二个典型的弊端是：稍微留心一点便发现 turn的值是交替改变的，无法满足线程 i 连续进入临区的需求。

思路二

如何改善算法一遗留的——同一个进程无法连续进入临界区的问题呢。

算法一是仅仅使用了一个变量 turn 来代表当前能够进入临界区的线程。那么可不可以这样想：使用个变量，每个变量对应一个线程，每个变量的意义是：0 - 我不能进入临界区；1- 我可以进入临界区。

伪代码思路：

```
/* 全局变量 */
int flag[2] = {0, 0}; // 一个flag数组，flag[0] 代表线程 i， flag[1] 代表线程 j
```

● 对于线程 i 而言

```
while(flag[1] == 1);
flag[0] = 1; // 表示 线程 i 已经进入临界区中
/* 临界区 */
do something
flag[0] = 0;
```

● 对于线程 j 而言

```
while(flag[0] == 1);
flag[1] = 1; // 表示线程 j 已经进入临界区中
/* 临界区 */
do something
flag[1] = 0;
```

这样通过多个变量，每个线程拥有一个属于自己的变量来记录自己的状态，便能够解决算法一的问题同一个进程连续两次进入临界区，而不必只能交替进入。

然而：算法二又引入了新的问题：**对于 flag标志的判断，可以出现在某一时刻，两个线程同时满足 flag[0] == 1 && flag[1] == 1 的条件，无法满足互斥条件。**

解决了思路一的问题，却引入了一个新的问题，那么又该如何解决思路二中无法满足互斥的问题呢?

思路三

思路二无法互斥的原因是：在while循环判断后 → flag标志 = 1被改变之前，其他线程是同样可以满足hile循环的判断， 并改变自身的flag标志 = 1。

我们可以把flag标志 = 1放到while循环的前面：**我们先改变了自身的标志，这样另一个线程就无法跳while循环的判断了**

- 对于线程 i 而言

```
flag[0] = 1; // 表示 线程 i 已经进入临界区中
while(flag[1] == 1);
/* 临界区 */
do something
flag[0] = 0;
```

- 对于线程 j 而言

```
flag[1] = 1; // 表示线程 j 已经进入临界区中
while(flag[0] == 1);
/* 临界区 */
do something
flag[1] = 0;
```

但很可惜的是：虽然这样确实解决了互斥的问题，但却引来了一个更为严重的问题——**死锁** ： 同时塞在while循环中，谁也出不去。

那出不去我该怎么办呢？就像两辆小汽车，过一条小巷子，谁也不让谁，干巴巴的等在这里，等到地天荒，海枯石烂，等到汽车都报废了，依然是谁也过不去。

想一下现实中遇到这种情形该如何？ 只容一辆车的巷子，只能做一人的座位，只能。。。 作为一名着红领巾的三好学生（好吃，好睡，好帅 ^ ^）遇到这种情况时：让人三尺又何妨嘛。

直接看伪代码：

- 对于线程 i 而言

```
flag[0] = 1; // 表示 线程 i 要进入临界区中
while(flag[1] == 1) {
    // flag[1] == 1 表明 flag[1] 代表的线程也要进入临界区
```

```
    flag[0] = 0; // 既然你要，我退让一步喽
    sleep();     // 回去睡个午觉再来
    flag[0] = 1; // 睡醒了，看看资源是不是还被占用了。
}
/* 临界区 */
do something
flag[0] = 0;
```

- 对于线程 j 而言

```
flag[1] = 1; // 表示 线程 j 要进入临界区中
while(flag[0] == 1) {
    // flag[0] == 1 表明 flag[0] 代表的线程也要进入临界区
    flag[1] = 0; // 既然你要，我退让一步喽
    sleep();     // 回去睡个午觉再来
    flag[1] = 1; // 睡醒了，看看资源是不是还被占用了。
}
/* 临界区 */
do something
flag[0] = 0;
```

让线程互相退让一步。然而这样依旧存在缺点： - - 两个人都互相谦让，结果都不要这个座位，等到分钟后，两人又同时回来，结果又互相谦让。

虽然理论上讲依然会造成多次循环，但只要二者的时间稍微错开一些，便会打破这种循环。严格意义来讲，我们不能把这种情况称为 **死锁**， 可以叫 活锁。

总结： 然而可惜的是，虽然算法三的互相谦让基本上解决了算法一和算法二中遗留的问题，但可惜是捉摸不定的**谦让算法** 天知道会浪费我多少CPU资源，浪费多少青春和时间。所以，还是拒绝了最后的**让算法**

Dekker 算法

我们如何避免两个线程互相永无止境的谦让呢？ 就像我们在公交车或地铁上让座的时候，会优先把座主动让给老爷爷或者十月怀胎的阿姨 （什么，让给身材暴露的妹纸，- - 禽兽）。我们线程也是一样我们可以提前设置两个线程谁是老爷爷（不过在程序里面老爷爷的角色是互相交替的， 今天你让我，次我让你，世界多美好）。

再次引入 turn 变量： **当且仅当** 两个线程**发生竞争条件**时，根据 turn 的值来**判断谁谦让谁。turn值轮流改变的**

注意 turn 的含义和算法一中 turn 的含义不同， 该turn的值代表的是 **谦让**。

伪代码思路

```
turn = j; // 提前设置：j 线程是老爷爷
```

- 对于线程 i 而言

```
flag[0] = 1;
while(flag[1] == 1) {
    // flag[1] 代表的 j 线程也想坐座位
    // 判断 j 是否是 老爷爷
    if (turn == j) {
```

```
        // j 是老爷爷，那么 i 则主动承让
        flag[0] = 0; // 退出资源争夺
        while (turn == j) ;
        flag[0] = 1;
    }
}

/* 临界区 */
turn = j; // 修改谦让线程
flag[0] = 0;
```

- 对于线程 j 而言

```
flag[1] = 1;
while(flag[0] == 1) {
    // flag[0] 代表的 i 线程也想坐座位
    // 判断 i 是否是 老爷爷 (turn 值是轮流改变的)
    if (turn == i) {
        // i 是老爷爷，那么 j 则主动承让
        flag[1] = 0; // 退出资源争夺
        while (turn == i) ;
        flag[1] = 1;
    }
}

/* 临界区 */
turn = i; // 修改谦让线程
flag[1] = 0;
```

再次总结： 我们从算法一到算法三的过程中

- （一）：可以互斥、却只能交替
- （二）：可以连续、却无法满足互斥
- （三）：可以连续、可以互斥、却产生死锁
- （三）：可以连续、可以互斥、没有死锁、却因活锁而带来极大的性能问题
- **到 Dekker** 解决了以上全部问题。

那么 Dekker的缺点是什么？？？

对于并发互斥问题的算法中，Dekker算法最大的缺点就是： **太复杂**。代码比较复杂，而且程序不太易证明其并发的正确性。试想当拥有多个进程的时候，该如何去控制这些变量之间的关系。

好吧，又一位伟大的数学家简化了上述的思想，那么下面我们来介绍一下最后一个基于软件实现锁的法思想。

Peterson 算法

Peterson 算法依旧是使用了 turn变量 和 flag数组。

回到线程三的思路中来：

在while循环判断前，可能发生两个线程同时进入判断从而造成死锁（无限循环）。

```
flag[0] = 1; // n 为线程 i / j 的id
while(flag[j] == 1);
```

为了解决这个问题：当两个flag标志同时成立时，我们可以多加一个判断条件 turn == 其他线程id。果 turn不等于其他线程id（即此时为自身线程id），那么表明我可以去访问临界区。如果turn等于其线程id，那么表明是其他线程可以访问临界区。

所以： 只要保证 **任意时刻，turn 的值是唯一的**

此时我们的代码变成了

- 对于线程 i 而言

```
flag[0] = 1;
turn = i;
while(flag[1] == 1 && turn == j);
/* 临界区 */
flag[0] = 0;
```

- 对于线程 j 而言

```
flag[1] = 1;
turn = j;
while(flag[0] == 1 && turn == i);
/* 临界区 */
flag[1] = 0;
```

而永远不会说存在一个时刻：使得 turn == j 成立，而且 turn == i 也成立。因为 turn = i 和 turn = j 的赋值，总是会一个值覆盖另一个值的。

总结： 算法 Peterson 比起 Dekker 更加简洁。

总结

本篇文章中，自己的思路是：

并发面临的问题是什么 → 如何解决并发面临的问题呢？（提出了锁） → 那么锁的实现原理又是什么？

关于锁的实现

花了很大一部分篇幅在讲锁的实现原理，个人不是太喜欢写下来答案，而是更偏向于展现一个人思考过程（自己的小小看法）。中间的算法分析都是先有了问题，然后找解决的办法。所以想体现出给大层层递进的感觉，可能无法写出自己想要的效果，欢迎大家热烈的批评和建议。 ^ ^ 欢迎欢迎，热欢迎。