- 线程间同步方式
  - 引言
  - 互斥锁
    - 探究底层，实现一个锁
      - 测试并加锁(TAS)
      - 比较并交换(CAS)
      - 另一个问题，过多的自旋？
    - 回到互斥锁
  - 信号量
    - 理解信号量
    - 有名信号量
    - 无名信号量
    - 总结
  - 条件变量
    - 什么是条件变量？
    - 相关函数
      - 1. 初始化
      - 2. 等待条件
      - 3. 通知条件
    - 用法与思考
  - 实践——读写者锁

文章已收录我的仓库：Java学习笔记与免费书籍分享

线程间同步方式

引言

不同线程间对临界区资源的访问可能会引起常见的并发问题，我们希望线程原子式的执行一系列指令，但由于单处理器上的中断，我们必须想一些其他办法以同步各线程，本文就来介绍一些线程间的同步方式。

互斥锁

互斥锁(又名互斥量)，强调的是资源的访问互斥：互斥锁是用在多线程多任务互斥的，当一个线程占用了某一个资源，那么别的线程就无法访问，直到这个线程unlock，其他的线程才开始可以利用这个资源。

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

注意理解trylock函数，这与普通的lock不一样，普通的lock函数在资源被锁住时会被堵塞，直到锁被释放。

trylock函数是非阻塞调用模式，也就是说如果互斥量没被锁住，trylock函数将把互斥量加锁，并获得对共享资源的访问权限; 如果互斥量被锁住了，trylock函数将不会阻塞等待而直接返回EBUSY，表示共享资源处于忙状态，这样就可以避免死锁或饿死等一些极端情况发生。

探究底层，实现一个锁

实现一个锁必须需要硬件的支持，因为我们必须要保证锁也是并发安全的，这就需要硬件支持以保证锁内部是原子实现的。

很容易想到维护一个全局变量flag，当该变量为0时，允许线程加锁，并设置flag为1；否则，线程必须挂起等待，直到flag为0.

```
typedef struct lock_t {
    int flag;
}lock_t;

void init(lock_t &mutex) {
    mutex->flag = 0;
}

void lock(lock_t &mutex) {
    while (mutex->flag == 1) {;} //自旋等待变量为0才可进入
    mutex->flag = 1;
}

void unlock(lock_t &mutex) {
    mutex->flag = 0;
}
```

这是基于软件的初步实现，初始化变量为0，线程自旋等待变量为0才可进入，这看上去似乎并没有什么毛病，但是仔细思考，这是有问题的：

Thread 1	Thread 2
call lock() while (flag == 1) interrupt: switch to Thread 2	

	call lock()
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1

flag = 1; // set flag to 1 (too!)	

当线程恰好通过while判定时陷入中断，此时并未设置flag为1，另一个线程闯入，此时flag仍然为0，通过while判定进入临界区，此时中断，回到原线程，原线程继续执行，也进入临界区，这就造成了同步问题。

在while循环中，仅仅设置mutex->flag == 1是不够的，尽管他是一个原语，我们必须有更多的代码，同时，当我们引入更多代码时，我们必须保证这些代码也是原子的，这就意味着我们需要硬件的支持。

我们思考上面代码为什么会失败？原因是当退出while循环时，在这一时刻flag仍然为0，这就给了其他线程抢入临界区的机会。

解决办法也很直观 —— 在退出while时，借助硬件支持保证flag被设置为1。

测试并加锁(TAS)

我们编写如下函数：

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr;
    *old_ptr = new;
    return old;
}
```

同时重新设置while循环：

```
void lock(lock_t &mutex) {
    while (TestAndSet(mutex->flag, 1) == 1) {;} //自旋等待变量为0才可进入
    mutex->flag = 1;
}
```

这里，我们借助硬件，保证TestAndSet函数是原子执行的，现在锁可以正确的使用了。当flag为0时，我们通过while测试时已经将flag设置为1了，其他线程已经无法进入临界区；如果flag = 1，我们还是将其设置为1(局限性)，并没有改变其值。

TAS的局限性在于它几乎只能用于二值锁的实现，诸如信号量的实现便无能为力了，但优点在于简单高效。

比较并交换(CAS)

我们编写如下函数：

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected) {
        *ptr = new;
    }
    return actual;
}
```

同样的，硬件也应该支持CAS原语以保证CAS内部也是安全的，现在重新设置while：

```
void lock(lock_t &mutex) {
    while (CompareAndSwap(mutex->flag， 0， 1) == 1) {;} //自旋等待变量为0
才可进入
    mutex->flag = 1;
}
```

现在锁可以正确的使用了，当flag为0时，我们通过while测试时已经将flag设置为1了，其他线程已经无法进入临界区。

此外你可能发现CAS所需要更多的寄存器，在将来研究`synchronozation`时，你会发现它的妙处。

另一个问题，过多的自旋？

你可能发现了，尽管一个线程未能获得锁，其仍然在不断while循环以占用CPU资源，一个办法就是当线程未能获得锁，进入休眠以释放CPU资源(条件变量)，当一个线程释放锁时，唤醒一个正在休眠的线程。不过这样也有缺点，进入休眠与唤醒一个锁也是需要时间的，当一个线程很快就能释放锁时，多等等是比陷入休眠更好的选择。

Linux下采用两阶段锁，第一阶段线程自旋一定时间或次数等待锁的释放，当达到一定时间或一定次数时，进入第二阶段，此时线程进入休眠。

回到互斥锁

互斥锁提供了并发安全的基本保证，互斥锁用于保证对临界区资源的安全访问，但何时需要访问资源并不是互斥锁应该考虑的事情，这可能是条件变量该考虑的事情。

如果线程频繁的加锁和解锁，效率是非常低效的，这也是我们必须要考虑到的一个点。

信号量

理解信号量

信号量并不用来传送资源，而是用来保护共享资源，理解这一点是很重要的，信号量 s 的表示的含义为*同时允许访问资源的最大线程数量*，它是一个全局变量。

信号量 s 是具有非负整数值的全局变量，由两种特殊的**原子操作**来实现，这两种原子操作称为 P 和 V ：

- P(s)：如果 s 的值大于零，就给它减1，然后立即返回，进程继续执行。；如果它的值为零，就挂起该进程的执行，等待 s 重新变为非零值。
- V(s)：V操作将 s 的值加1，如果有任何进程在等在 s 值变为非0，那么V操作会重启这些等待进程中的其中一个(随机地)，然后由该进程执行P操作将 s 重新置为0，而其他等待进程将会继续等待。

在进程中也可以使用信号量，对于信号量的理解进程中与线程中并无太大差异，都是用来保护资源，关于更多信号量的理解参见这篇文章: JavaLearningNotes/进程间通信方式。

有名信号量

有名信号量以文件的形式存在，即时是不同进程间的线程也可以访问该信号量，因此可以用于不同进程间的多线程间的互斥与同步。

创建打开有名信号量

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int
value);
//成功返回信号量指针；失败返回SEM_FAILED，设置errno
```

name是文件路径名，value设置为信号量的初始值。

关闭信号量，进程终止时，会调用它

```
int sem_close(sem_t *sem);      //成功返回0；失败返回-1，设置errno
```

删除信号量，立即删除信号量名字，当其他进程都关闭它时，销毁它

```
int sem_unlink(const char *name);
```

等待信号量，测试信号量的值，如果其值小于或等于0，那么就等待（阻塞）；一旦其值变为大于0就将它减1，并返回

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
//成功返回0；失败返回-1，设置errno
```

当信号量的值为0时，sem_trywait立即返回，设置errno为EAGAIN。如果被某个信号中断，sem_wait会过早地返回，设置errno为EINTR

发出信号量，给它的值加1，然后唤醒正在等待该信号量的进程或线程

```
int sem_post(sem_t *sem);
```

成功返回0；失败返回-1，不会改变它的值，设置errno，该函数是异步信号安全的，可以在信号处理程序里调用它

无名信号量

无名信号量存在于进程内的虚拟空间中，对于其他进程是不可见的，因此无名信号量用于一个进程体内各线程间的互斥和同步,使用如下API：

（1）sem_init 功能：用于创建一个信号量，并初始化信号量的值。 函数原型：

```
int sem_init (sem_t* sem, int pshared, unsigned int value);
```

函数传入值： sem:信号量。pshared:决定信号量能否在几个进程间共享。由于目前LINUX还没有实现进程间共享信息量，所以这个值只能取0。

（2）其他函数

```
int sem_wait       (sem_t* sem);
int sem_trywait   (sem_t* sem);
int sem_post       (sem_t* sem);
int sem_getvalue (sem_t* sem);
int sem_destroy   (sem_t* sem);
```

功能：

sem_wait和sem_trywait相当于P操作，它们都能将信号量的值减一，两者的区别在于若信号量的值小于零时，sem_wait将会阻塞进程，而sem_trywait则会立即返回。

sem_post相当于V操作，它将信号量的值加一，同时发出唤醒的信号给等待的线程。

sem_getvalue 得到信号量的值。

sem_destroy 摧毁信号量。

如果某个基于内存的信号量是在不同进程间同步的，该信号灯必须存放在共享内存区中，这要只要该共享内存区存在，该信号灯就存在。

总结

无名信号量存在于内存中，有名信号量是存在于磁盘上的，因此无名信号量的速度更快，但只适用于一个独立进程内的各线程；有名信号量可以速度欠缺，但可以使不同进程间的线程同步，这是通过共享内存实现的，共享内存是进程间的一种通信方式。

你可能发现了，当信号量的值s为1时，信号量的作用于互斥锁的作用是一样的，互斥锁只能允许一个线程进入临界区，而信号量允许更多的线程进入临界区，这取决于信号量的值为多少。

条件变量

什么是条件变量？

在互斥锁中，线程等待flag为0才能进入临界区；信号量中P操作也要等待s不为0…在多线程中，一个线程等待某个条件是很常见的，互斥锁实现一节中，我们采用自旋来实现，但这效率非常低效，是否有一个更专门、更高效的方式实现条件的等待？

它就是条件变量！条件变量(condition variable)是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待某个条件为真，而将自己挂起；另一个线程设置条件为真，并通知等待的线程继续。

由于某个条件是全局变量，因此**条件变量常使用互斥锁以保护**(这是必须的，是被强制要求的)。

条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定的条件发生。

线程可以使用条件变量来等待某个条件为真，注意理解并不是等待条件变量为真，条件变量(cond)是在多线程程序中用来实现"等待->唤醒"逻辑常用的方法，用于维护一个条件(这与条件变量是不同的概念)，线程用条件变量的用以等待条件成立，并不是说等待条件变量为真或为假，而是利用条件变量去等待某条件。条件变量是一个显式的队列，当条件不满足时，线程将自己加入等待队列，同时释放持有的互斥锁；当一个线程改变条件时，可以唤醒一个或多个等待线程(注意此时条件不一定为真)。

在条件变量上有两种基本操作：

- 等待（wait）：一个线程处于等待队列中休眠，此时线程不会占用互斥量，当线程被唤醒后，重新获得互斥锁(可能是多个线程竞争)，并重新获得互斥量。
- 通知（signal/notify）：当条件更改时，另一个线程发送通知以唤醒等待队列中的线程。

相关函数

1. 初始化

条件变量采用的数据类型是pthread_cond_t,，在使用之前必须要进行初始化,，这包括两种方式:

静态: 直接设置条件变量cond为常量PTHREAD_COND_INITIALIZER。

动态: pthread_cond_init函数, 是释放动态条件变量的内存空间之前, 要用pthread_cond_destroy对其进行清理。

```
int pthread_cond_init(pthread_cond_t *restrict cond, pthread_condattr_t
*restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
//成功则返回0, 出错则返回错误编号.
```

注意：条件变量占用的空间并未被释放。

cond：要初始化的条件变量；attr：一般为NULL。

2. 等待条件

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t
*restric mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t
*restrict mutex, const struct timespec *restrict timeout);
//成功则返回0, 出错则返回错误编号.
```

这两个函数分别是阻塞等待和超时等待，堵塞等到进入等待队列休眠直到条件修改而被唤醒；超时等待在休眠一定时间后自动醒来。

进入等待时线程释放互斥锁，而在被唤醒时线程重新获得锁。

3. 通知条件

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
//成功则返回0, 出错则返回错误编号.
```

这两个函数用于通知线程条件已被修改，调用这两个函数向线程或条件发送信号。

用法与思考

条件变量用法模板：

```
pthread_cond_t cond;  //条件变量
mutex_t mutex;  //互斥锁
```

```
int flag; //条件

//A线程
void threadA() {
    Pthread_mutex_lock(&mutex);  //保护临界资源，因为线程会修改全局条件flag
    while (flag == 1) //等待某条件成立
        Pthread_cond_wait(&cond, &mutex);  //不成立则加入队列休眠，并释放锁
    ....dosomthing // 条件成立
    ....change flag   //条件被修改
    Pthread_cond_signal(&cond); //发送信号通知条件被修改
    Pthread_mutex_unlock(&mutex); //发送信号后尽量快速释放锁，因为被唤醒的线程会尝试获得锁
}


//B线程
void threadB() {
    Pthread_mutex_lock(&mutex);  //保护临界资源
    while (flag == 0) //等待某条件成立
        Pthread_cond_wait(&cond, &mutex);  //不成立则加入队列休眠，并释放锁
    ....dosomthing // 条件成立
    ....change flag   //条件被修改
    Pthread_cond_signal(&cond); //放松信号后尽量快速释放锁，因为被唤醒的线程会尝试获得锁
    Pthread_mutex_unlock(&mutex);
}
```

通过上面的一个例子，应该很好理解条件变量与条件的区别，条件变量是一个逻辑，它并不是while循环里的bool语句，我相信很多初学者都有这么一个误区，即条件变量就是线程需要等待的条件。条件是条件，线程等待条件而不是等待条件变量，条件变量使得线程更高效的等待条件成立，是一组等待 — 唤醒 的逻辑。

注意这里仍然要使用while循环等待条件，你可能会认为明明已经上锁了别的线程无法强入。事实上当线程A陷入休眠时会释放锁，而当其被唤醒时，会尝试获得锁，而正在其尝试获得锁时，另一个线程B现在尝试获得锁，并且抢到锁进入临界区，然后修改条件，使得线程A的条件不再成立，线程B返回，此时线程A终于获得锁了，并进入临界区，但此时线程A的条件根本已经不成立，他不该进入临界区！

此外，被唤醒也不代表条件成立了，线程陷入休眠时可能会因为超时而返回，这种情况下条件并不会成立；此外上述代码线程B修改flag = 3，并且唤醒线程A，这里线程A的条件根本不符合，所以必须重复判定条件(这是糟糕的情况)。互斥锁和条件变量的例子告诉我们：**在等待条件时，总是使用while而不是if！**

陷入休眠的线程必须释放锁也是有意义的，如果不释放锁，其他线程根本无法修改条件，休眠的线程永远都不会醒过来！

实践——读写者锁

读取锁——共享；写入锁——独占。即：读线程可以加多个，而写线程只能有一个，并且读者和写者不能同时工作。

这种情况下由于允许多个读者共享临界区效率会高效，我们来考虑实现的问题：只允许一个写者工作，那么一定需要一个互斥量或二值信号量来维护，我们称为写者锁；由于读者和写者不能同时工作，第一个读者必须尝试获取写者锁，而一旦读者数量大于1，则后续读者无须尝试获取写者锁而可直接进入，注意到这里存在全局读者数量变量，因此读者也需要一个锁以维护全局读者数量，最后一个退出的读者必须负责释放读者锁。

知晓原理，快去自己动手实现一个读写者锁把！

Linux下通过`pthread_rwlock`函数族实现。

其核心思想便是如此。