

线程安全:

多个执行流对临界资源进行争抢访问,而不会造成数据二义或者逻辑混乱;称这段争抢访问的过程是线程安全的;线程安全的实现:如何保证多个执行流对临界资源进行争抢访问而不会造成数据二义

同步:通过条件判断,实现对临界资源访问的时序合理性

互斥:通过唯一访问,实现对临界资源访问的安全性

互斥的实现技术:互斥锁/信号量

实现互斥的原理:只要保证同一时间只有一个执行流能够访问资源就是互斥

对临界资源进行状态标记:没人访问的时候标记为1,表示可访问;有人正在访问的时候,就标记为0,表示不可访问;在对临界资源进行访问之前先进行状态的判断,决定是否能够访问,不能访问则使其休眠

能够实现互斥的技术中:互斥锁

互斥锁:其实就是一个计数器,只有0/1的计数器,用于标记资源当前的访问状态 1-可访问0-不可访问

互斥锁想要实现互斥,每个线程在访问临界资源之前都要先访问同一个互斥锁(加锁);

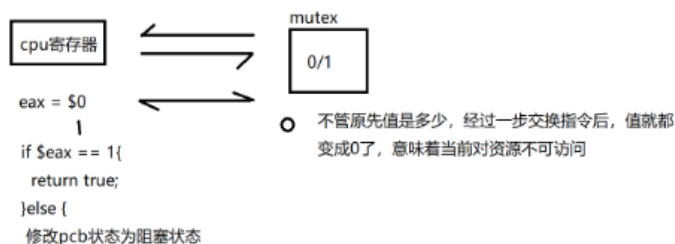
意味着互斥锁本身就是一个临界资源,(涉及到计数器的修改修改过程必须保证安全,因为如果连自己都保护不好,就不能保护别人)

互斥锁的计数器操作如何实现原子性:

1.将cpu寄存器上的值修改为0,然后与内存中计数器进行数据交换(意味着这时候计数器变程了0,谁来访问,都是-种不可访问状态,别人都进不去,这时候,寄存器就可以慢慢判断是都可以访问了)

2.若寄存器交换后数据为0,则表示当前不可访问,则将pcb状态置为阻塞状态,线程将被挂起等待,若寄存器交换后数据为1,则表示当前可以访问,则加锁操作直接返回,表示加锁成功--继续可以访问资源

3.访问完数据之后,要进行解锁操作(将内存中计数器的值再修改回来)



1.互斥锁是一个计数器,本身的计数操作是原子性(如何保证自己的原子性)

2.互斥锁如何实现互斥,通过在访问临界资源之前先加锁访问互斥锁,来进行状态判断是否可加锁

互斥锁的代码操作流程:

1.定义互斥锁变量pthread_mutex_t mutex;

2.初始化互斥锁

mutex=PTHREAD_MUTEX_INITIALIZER

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)

mutex:互斥锁变量首地址

attr:互斥锁属性---通常置NULL,

3.在对临界资源访问之前,先加锁(访问锁,判断是否可以访问) --- 保护对临界资源访问的过程

int pthread_mutex_lock(pthread_mutex_t *mutex);阻塞加锁---如果不能加锁,则一直等待

int pthread_mutex_trylock(pthread_mutex_t *mutex);非阻塞加锁---如果不能加锁,则立即报错返回,若可以加锁,则加锁后返回

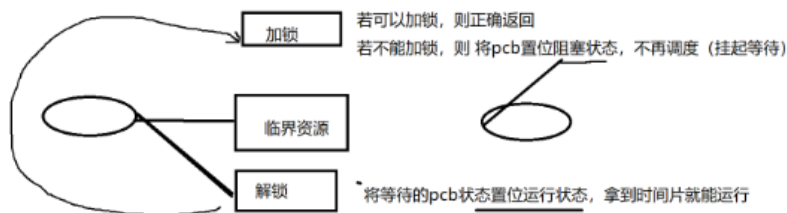
4.在对临界资源访问完毕之后,记得解锁(把状态标记为可访问)

int pthread_mutex_unlock(pthread_mutex_t *mutex);

5.不使用锁了,最终要释放资源,销毁互斥锁

int pthread_mutex_destroy(pthread_mutex_t *mutex);

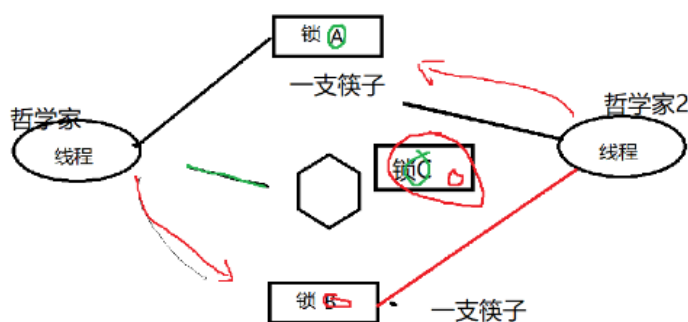
在黄牛抢票的例子中，加锁后，一个黄牛抢完了所有的票没有票了,为什么大家不退出



互斥锁的操作流程中, 需要注意的:

- 1.加锁后, 在任意有可能退出线程的地方记得解锁
- 2.锁的初始化一定要在创建线程之前,
- 3.锁的销毁一定是保证没有人使用互斥锁的时候

死锁:多个执行流对锁资源进行争抢访问, 但是因为推进顺序不当, 而导致互相等待;最终造成程序流程无法继续的情况



死锁产生的必要条件:必须具备的条件, 如果不具备就无法造成死锁--知道了必要条件, 就可以实现预防以及避免

- 1.互斥条件 一个锁不能大家同时加,我加了锁,别人就不能再加了----同一时间只有一个线程能够加锁
- 2.不可剥夺条件 我加的锁, 别人不能解, 只有我能解锁 线程加的锁, 只有自己能解
- 3.请求与保持条件 吃着碗里的,看着锅里的; 抢到了锁A,然后去抢锁B,但是抢不到锁B,也不释放锁A
- 4.环路等待条件 线程1抢到了锁A,然后去抢锁B,但是抢不到锁B ;线程2抢到了锁B,然后去抢锁A;但是抢不到锁A

如何预防死锁:防患于未然----在编写代码过程中注意破坏产生的必要条件即可

如何避免产生死锁:死锁检测算法/银行家算法---- 课后调研

银行家算法:将系统的运行分成了两种状态---安全/不安全

- 1.当前都有哪些资源 2.哪些资源已经分配给了谁 3.现在谁都想要要获取哪些资源

若给你分配你想要的资源, 是否会造成系统处于不安全状态--- 分配给一个线程想要的锁资源, 是否会造成环路等待,如果有可能造成, 当前就是不安全的, 则不能分配(回溯,将已经加的锁考虑是否释放掉)

加锁的时候采用非阻塞加锁方式/判断安全状态

破坏请求与保持条件:若不能加锁, 则将已经加的锁释放掉

按序加锁

同步的实现: 条件变量实现的

条件变量: 实现同步的思路向用户提供两个接口(一个是让线程陷入阻塞休眠的接口; 一个是唤醒线程休眠的接口) +pcb等待队列

同步: 通过条件判断 (什么时候能够访问资源, 什么时候不能访问; 若不能访问就要使线程阻塞; 若能访问了就要唤醒线程), 实现线程对临界资源访问的合理性

等待队列

若碗里有饭, 才能吃, 碗里没饭, 则不能吃, 因此要等待

顾客

碗是否为空

碗

碗是否为空

厨师

若碗里有饭, 则表示没人吃, 则不能做饭, 就要等待, 等到饭被吃了碗空了才能做饭

吃碗面, 唤醒厨师

唤醒等待的顾客, 来吃饭

条件变量: 只是向外提供了等待与唤醒的接口, 却没有提供条件判断(条件变量本身并不具备判断什么时候该等待, 什么时候该唤醒)的功能; 意味着, 条件判断需要用户自己来完成

条件变量提供的接口功能:

1. 定义条件变量 `pthread_cond_t cond`

2. 初始化条件变量

`pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);` // `cond=PTHREAD_COND_INITIALIZER`

3. 一个线程等待的接口: `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`

条件变量是搭配互斥锁使用的: 条件变量并不提供条件判断的功能, 需要用户去判断(通常条件的判断是一个临界资源的访问); 因此这个临界资源的访问, 就需要受保护, 使用互斥锁保护。

`pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *abstime);`

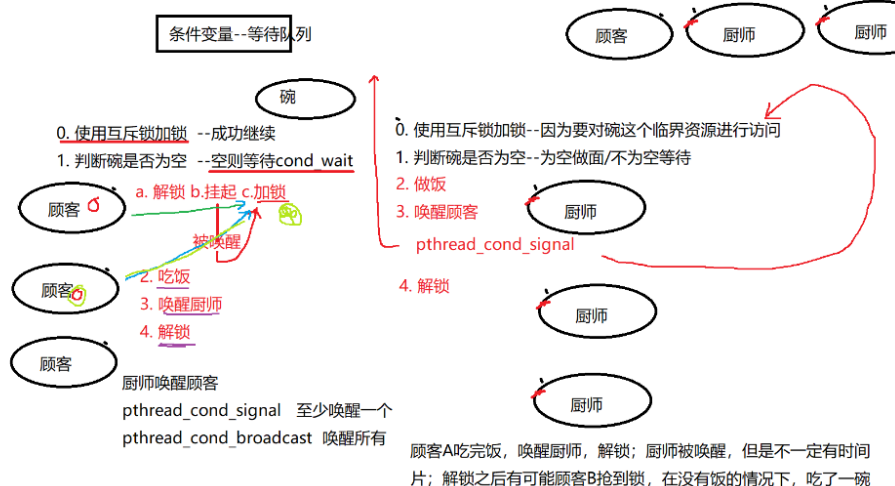
限制等待时长的阻塞操作: 等待一段指定的时间, 时间到了调用就会报错返回-- `ETIMEDOUT`

4. 一个唤醒线程的接口:

`pthread_cond_signal(pthread_cond_t *cond);` // 唤醒至少一个等待的线程

`pthread_cond_broadcast(pthread_cond_t *cond);` // 唤醒所有等待的线程

5. 若不使用条件了则销毁释放资源: `pthread_cond_destroy(pthread_cond_t *cond);`



因此条件的判断应该是一个循环判断, 有多个顾客都被唤醒的情况下, 大家争抢锁, 拿到锁之后再重新判断一次有没有饭, 有饭则不再等待, 下去吃饭

没有饭, 会重新调用 `pthread_cond_wait` 陷入休眠 避免在没有饭的情况下出现吃饭的情况

为什么只有一个顾客和厨师的时候好好的, 顾客和厨师多了之后出现问题了??

只有一个顾客和厨师的时候: 一个做好了, 另一个才能吃, 一个吃完了, 另一个才能做

多了之后就会出问题:

循环判断之后, 程序逻辑正确了, 做一碗, 吃一碗, 吃一碗, 做一碗; 但是程序会卡死, 为什么呢?

因为pcb等待队列中, 既有顾客pcb也有厨师pcb; 顾客被唤醒吃饭之后, 各科要去队列中唤醒厨师; 但是因为大家都在同一个队列中, 因此有肯能顾客唤醒的不是厨师, 而还是另一个顾客, 因此就导致了因为唤醒的角色错误, 顾客又因为没有饭而陷入休眠

所有的pcb加入到同一个队列中有可能出现唤醒角色错误的情况, 因此因该不同的角色等待在不同的队列上, 这样唤醒的时候, 就不会唤醒错了

所以在程序中角色有多少种，条件变量就应该有多少个(每种角色等待在不同的条件变量等待队列中)

条件变量:

条件变量实现同步的原理:两个接口+pcb等待队列

操作流程:用户不能访问资源的时候调用接口陷入等待，其它线程产生资源，然后调用接口唤醒等待队列中的线程;是否能够访问的条件判断需要用户自己完成，并且需要互斥保护

- 1.加锁
- 2.用户自己进行条件判断，不能访问，则调用pthread_cond_wait陷入等待
- 3.被唤醒之后能够访问，则访问数据，获取资源
- 4.唤醒生产资源的线程
- 5.解锁

注意实现:细节问题

1. pthread_cond_wait中包含了三步操作(解锁+休眠+被唤醒后加锁/并且解锁和休眠是一个原子操作)
- 2.用户自己进行的条件判断需要使用while循环判断 (唤醒两个顾客，一个吃面，一个卡在锁上，吃完面的，解锁之后,应该厨师做面，但是解锁，谁都有可能抢到，因此另一个顾客会有可能在没有面的情况下抢到锁吃面)
- 3.不同的角色应该等待在不同的条件变量上(有多少角色，就有多少条件变量)，做到唤醒分明:厨师唤醒顾客队列:顾客唤醒厨师队列

复习:

互斥如何实现:互斥锁

互斥锁:

本质:也就是一个只有0/1的计数器;通过这个0/1标记临界资源当前的访问状态;

在访问临界资源之前，可以先去访问互斥锁计数器，判断是否可以访问;若可以访问，则在访问临界资源之前，将状态标记为不可访问(别人来访问的话就会看到当前是一个不可访问状态);并且互斥锁计数器的改变本身也是一个原子操作;

互斥锁本身计数的安全实现:通过寄存器与内存之间的一次数据交换(原子性不可打断)，实现计数器的数据改变;保证自己的安全

若不能访问临界资源，则需要将当前线程置为阻塞状态，不再调度(让线程挂起等待);等到互斥锁解锁之后，会将所有等待线程的状态置为运行状态，拿到时间片就可以开始运行了;

互斥锁的操作流程:

- 1.定义互斥锁变量pthread_mutex_t mutex;
- 2.初始化互斥锁
mutex=PTHREAD_MUTEX_INITIALIZER; pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- 3.对临界资源访问之前进行加锁操作
pthread_mutex_lock(pthread_mutex_t *mutex) / pthread_mutex_trylock(pthread_mutex_t *mutex)
- 4.对临界资源访问完毕之后进行解锁操作 pthread_mutex_unlock(pthread_mutex_t *mutex);
- 5.若不再使用互斥锁了记得释放互斥锁 pthread_mutex_destroy(pthread_mutex_t *mutex);

注意事项:

- 1.加锁保护的区域，最好只有对临界资源访问的过程---因为保护的越多，执行的所需时间越长，降低效率
- 2.加锁之后，在任意有可能退出线程的地方都要进行解锁操作; --- 若没有解锁直接退出，有可能造成其它获取所锁的线程卡死

1.线程等待 2.线程分离/ 3.线程安全概念/4.实现方式/5.如何实现互斥-互斥锁

