12.5 防止死锁的加锁机制¶

问题¶

你正在写一个多线程程序，其中线程需要一次获取多个锁，此时如何避免死锁问题。

解决方案¶

在多线程程序中，死锁问题很大一部分是由于线程同时获取多个锁造成的。举个例子：一个线程获取了第一个锁，然后在获取第二个锁的 时候发生阻塞，那么这个线程就可能阻塞其他线程的执行，从而导致整个程序假死。 解决死锁问题的一种方案是为程序中的每一个锁分配一个唯一的id，然后只允许按照升序规则来使用多个锁，这个规则使用上下文管理器 是非常容易实现的，示例如下：

```
import threading
from contextlib import contextmanager

# Thread-local state to stored information on locks already acquired
_local = threading.local()

@contextmanager
def acquire(*locks):
    # Sort locks by object identifier
    locks = sorted(locks, key=lambda x: id(x))

    # Make sure lock order of previously acquired locks is not violated
    acquired = getattr(_local,'acquired',[])
    if acquired and max(id(lock) for lock in acquired) >= id(locks[0]):
        raise RuntimeError('Lock Order Violation')

    # Acquire all of the locks
    acquired.extend(locks)
    _local.acquired = acquired

    try:
        for lock in locks:
            lock.acquire()
        yield
    finally:
        # Release locks in reverse order of acquisition
        for lock in reversed(locks):
            lock.release()
        del acquired[-len(locks):]
```

如何使用这个上下文管理器呢？你可以按照正常途径创建一个锁对象，但不论是单个锁还是多个锁中都使用 `acquire()` 函数来申请锁， 示例如下：

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print('Thread-1')
```

```
def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果你执行这段代码，你会发现它即使在不同的函数中以不同的顺序获取锁也没有发生死锁。 其关键在于，在第一段代码中，我们对这些锁进行了排序。通过排序，使得不管用户以什么样的顺序来请求锁，这些锁都会按照固定的顺序被获取。 如果有多个 `acquire()` 操作被嵌套调用，可以通过线程本地存储（TLS）来检测潜在的死锁问题。 假设你的代码是这样写的：

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():

    while True:
        with acquire(x_lock):
            with acquire(y_lock):
                print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock):
            with acquire(x_lock):
                print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果你运行这个版本的代码，必定会有一个线程发生崩溃，异常信息可能像这样：

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/local/lib/python3.3/threading.py", line 639, in _bootstrap_inner
    self.run()
  File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
  File "deadlock.py", line 49, in thread_1
    with acquire(y_lock):
  File "/usr/local/lib/python3.3/contextlib.py", line 48, in __enter__
    return next(self.gen)
  File "deadlock.py", line 15, in acquire
```

```
    raise RuntimeError("Lock Order Violation")
RuntimeError: Lock Order Violation
>>>
```

发生崩溃的原因在于，每个线程都记录着自己已经获取到的锁。 `acquire()` 函数会检查之前已经获取的锁列表， 由于锁是按照升序排列获取的，所以函数会认为之前已获取的锁的id必定小于新申请到的锁，这时就会触发异常。

讨论¶

死锁是每一个多线程程序都会面临的一个问题（就像它是每一本操作系统课本的共同话题一样）。根据经验来讲，尽可能保证每一个 线程只能同时保持一个锁，这样程序就不会被死锁问题所困扰。一旦有线程同时申请多个锁，一切就不可预料了。

死锁的检测与恢复是一个几乎没有优雅的解决方案的扩展话题。一个比较常用的死锁检测与恢复的方案是引入看门狗计数器。当线程正常 运行的时候会每隔一段时间重置计数器，在没有发生死锁的情况下，一切都正常进行。一旦发生死锁，由于无法重置计数器导致定时器 超时，这时程序会通过重启自身恢复到正常状态。

避免死锁是另外一种解决死锁问题的方式，在进程获取锁的时候会严格按照对象id升序排列获取，经过数学证明，这样保证程序不会进入 死锁状态。证明就留给读者作为练习了。避免死锁的主要思想是，单纯地按照对象id递增的顺序加锁不会产生循环依赖，而循环依赖是 死锁的一个必要条件，从而避免程序进入死锁状态。

下面以一个关于线程死锁的经典问题：“哲学家就餐问题”，作为本节最后一个例子。题目是这样的：五位哲学家围坐在一张桌子前，每个人 面前有一碗饭和一只筷子。在这里每个哲学家可以看做是一个独立的线程，而每只筷子可以看做是一个锁。每个哲学家可以处在静坐、 思考、吃饭三种状态中的一个。需要注意的是，每个哲学家吃饭是需要两只筷子的，这样问题就来了：如果每个哲学家都拿起自己左边的筷子， 那么他们五个都只能拿着一只筷子坐在那儿，直到饿死。此时他们就进入了死锁状态。 下面是一个简单的使用死锁避免机制解决“哲学家就餐问题”的实现：

```
import threading

# The philosopher thread
def philosopher(left, right):
    while True:
        with acquire(left,right):
            print(threading.currentThread(), 'eating')

# The chopsticks (represented by locks)
NSTICKS = 5
chopsticks = [threading.Lock() for n in range(NSTICKS)]

# Create all of the philosophers
for n in range(NSTICKS):
    t = threading.Thread(target=philosopher,
                         args=(chopsticks[n],chopsticks[(n+1) % NSTICKS]))
    t.start()
```

最后，要特别注意到，为了避免死锁，所有的加锁操作必须使用 `acquire()` 函数。如果代码中的某部分绕过acquire 函数直接申请锁，那么整个死锁避免机制就不起作用了。