522031910747+李若彬+HW10

Part 1: Deadlock与Livelock

> 请参考课件并查阅相关资料，分别解释deadlock和livelock并举例说明。

Deadlock（死锁）

定义:
当两个以上的运算单元，双方都在等待对方停止执行，以获取系统资源，但是没有一方提前退出时，就称为死锁。

死锁的四个条件是:

- **禁止抢占**（no preemption）：系统资源不能被强制从一个进程中退出。
- **持有和等待**（hold and wait）：一个进程可以在等待时持有系统资源。
- **互斥**（mutual exclusion）：资源只能同时分配给一个行程，无法多个行程共享。
- **循环等待**（circular waiting）：一系列进程互相持有其他进程所需要的资源。

死锁只有在四个条件同时满足时发生，预防死锁必须至少破坏其中一项。

示例:
假设有两个进程P1和P2，以及两个资源R1和R2。死锁的发生过程如下：

1. 进程P1已经获取了资源R1，并请求资源R2。
2. 进程P2已经获取了资源R2，并请求资源R1。
3. 由于资源R1已经被P1占用，而资源R2已经被P2占用，导致P1和P2都在等待对方释放资源，从而进入死锁状态。

图示:

```
P1 -> R1 -> P2 -> R2 -> P1
```

Livelock（活锁）

定义:

活锁指的是任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。一组进程不断改变状态以响应彼此的动作，但没有任何进展。与死锁不同，活锁中的进程并不是静止不动，而是不断尝试操作但始终无法成功完成任务。

示例:
假设有两个进程P1和P2，它们需要同时获取两个资源R1和R2才能继续执行。活锁的发生过程如下：

1. P1尝试获取R1，成功。
2. P2尝试获取R2，成功。
3. P1发现还需要R2，等待释放R2。
4. P2发现还需要R1，等待释放R1。
5. P1检测到P2在等待，于是释放R1并重新尝试获取R1和R2。
6. P2检测到P1在等待，于是释放R2并重新尝试获取R1和R2。
7. 这种情况不断重复，但P1和P2始终无法同时获取R1和R2，导致活锁。

图示:

```
P1: release R1 -> try R1 and R2 ->
P2: release R2 -> try R1 and R2 ->
(repeat)
```

总结

- **死锁**：进程相互等待对方释放资源，导致所有进程都无法继续执行，形成僵局。
- **活锁**：进程不断改变状态尝试获取资源，但由于相互影响，始终无法成功完成任务，形成持续循环。

Part 2: 并发栈

> 并发栈，指的是支持被多个线程同时进行操作，且能保证线程安全的栈。请参考课件中的并发队列，思考如何设计并发栈（但不要求实现）。具体的，请回答以下问题：
>
> 1. 你将采用什么方式，使得多个线程可以同时进行push、pop和isEmpty?
> 2. 你的设计中，可能存在的性能瓶颈在哪?
> 3. 你的设计中是否会存在deadlock或者livelock?

1. 你将采用什么方式，使得多个线程可以同时进行push、pop和isEmpty?

选择锁机制或无锁算法:

- **锁机制**：可以使用细粒度的锁，例如每个操作（push, pop, isEmpty）使用独立的锁，以减少线程之间的争用。
  - `push`和`pop`可以使用同一个锁，这样可以确保在一个线程执行push或pop时，另一个线程不会修改栈的状态。
  - `isEmpty`可以使用读写锁，在没有线程进行push或pop操作时，多个线程可以同时检查栈是否为空。
- **无锁算法**：可以使用原子操作操作实现（CAS）。

- 使用一个原子变量来指向栈顶元素，利用CAS操作来实现push和pop操作。将几步操作结合起来一起执行。

2. 你的设计中，可能存在的性能瓶颈在哪?

锁机制:

- **锁竞争**：多个线程同时尝试获取相同的锁，可能导致性能下降。尤其是在高并发环境下，锁竞争会变得更加明显。
- **锁的开销**：频繁的锁获取和释放会增加系统开销。

无锁算法:

- **CAS操作失败重试**：在高并发情况下，CAS操作可能会频繁失败，导致大量的重试（重新读取在另一线程操作下改变的变量值），进而影响性能。

3. 你的设计中是否会存在deadlock或者livelock?

锁机制:

- **死锁**：如果使用不当，可能会引入死锁。例如，如果在push和pop操作中使用不同的锁，且这些锁的获取顺序不一致，就有可能导致死锁。
- **活锁**：在某些情况下，如果线程在检测到锁被占用时采取的操作会导致其他线程的相似反应，可能会出现活锁。

无锁算法:

- **死锁**：无锁算法设计良好通常不会出现死锁，因为没有显式的锁存在。
  - 无锁算法通过以下方式避免这些条件：
    - **互斥条件**：无锁算法不使用互斥锁，而是依赖原子操作来更新共享资源。
    - **保持和等待条件**：线程在进行操作时，不会等待其他线程持有的资源，而是重试或回退操作。
    - **不剥夺条件**：无锁算法允许线程在需要时重新尝试操作，而不是持有资源直到操作完成。
    - **循环等待条件**：由于没有显式锁和资源等待链，无锁算法不会形成循环等待。
- **活锁**：有可能出现活锁，特别是在高并发情况下，如果多个线程不断失败并重试CAS操作，可能会导致进程的进展非常缓慢甚至停滞。