加餐 - 自旋锁

概述

自旋锁是一种多线程同步机制，用于保护共享资源免受并发访问的影响。在多个线程尝试获取锁时，它们会持续自旋（即在一个循环中不断检查锁是否可用）而不是立即进入休眠状态等待锁的释放。这种机制减少了线程切换的开销，适用于短时间内锁的竞争情况。但是不合理的使用，可能会造成 CPU 的浪费。

原理

自旋锁通常使用一个共享的标志位（如一个布尔值）来表示锁的状态。当标志位为 true 时，表示锁已被某个线程占用；当标志位为 false 时，表示锁可用。当一个线程尝试获取自旋锁时，它会不断检查标志位：

- 如果标志位为 false，表示锁可用，线程将设置标志位为 true，表示自己占用了锁，并进入临界区。
- 如果标志位为 true（即锁已被其他线程占用），线程会在一个循环中不断自旋等待，直到锁被释放。

优点与缺点

优点

1. **低延迟**：自旋锁适用于短时间内的锁竞争情况，因为它不会让线程进入休眠状态，从而避免了线程切换的开销，提高了锁操作的效率。
2. **减少系统调度开销**：等待锁的线程不会被阻塞，不需要上下文切换，从而减少了系统调度的开销。

缺点

1. **CPU 资源浪费**：如果锁的持有时间较长，等待获取锁的线程会一直循环等待，导致 CPU 资源的浪费。
2. **可能引起活锁**：当多个线程同时自旋等待同一个锁时，如果没有适当的退避策略，可能会导致所有线程都在不断检查锁状态而无法进入临界区，形成活锁。

使用场景

1. **短暂等待的情况**：适用于锁被占用时间很短的场景，如多线程对共享数据进行简单的读写操作。
2. **多线程锁使用**：通常用于系统底层，同步多个 CPU 对共享资源的访问。

纯软件自旋锁类似的原理实现

自旋锁的实现通常使用原子操作来保证操作的原子性，常用的软件实现方式是通过 CAS（Compare-And-Swap）指令实现。以下是一个简单的自旋锁实现示例（伪代码）：

```
C++
#include <stdio.h>
#include <stdatomic.h>
#include <pthread.h>
#include <unistd.h>

// 使用原子标志来模拟自旋锁
atomic_flag spinlock = ATOMIC_FLAG_INIT;  // ATOMIC_FLAG_INIT 是 0

// 尝试获取锁
void spinlock_lock() {
    while (atomic_flag_test_and_set(&spinlock)) {
        // 如果锁被占用，则忙等待
    }
}

// 释放锁
void spinlock_unlock() {
    atomic_flag_clear(&spinlock);
}
```

```
C++
typedef _Atomic struct
{
#if __GCC_ATOMIC_TEST_AND_SET_TRUEVAL == 1
  _Bool __val;
#else
  unsigned char __val;
#endif
```

```
} atomic_flag;
```

- 功能描述

`atomic_flag_test_and_set` 函数检查 `atomic_flag` 的当前状态。如果 `atomic_flag` 之前没有被设置过（即其值为 false 或“未设置”状态），则函数会将其设置为 true（或“设置”状态），并返回先前的值（在这种情况下为 false）。如果 `atomic_flag` 之前已经被设置过（即其值为 true），则函数不会改变其状态，但会返回 true。

- 原子性

这个操作是原子的，意味着在多线程环境中，它保证了对 `atomic_flag` 的读取和修改是不可分割的。当一个线程调用此函数时，其他线程无法看到这个操作的任何中间状态，这确保了操作的线程安全性。

Linux 提供的自旋锁系统调用

```
C++
 #include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

注意事项

- 在使用自旋锁时，需要确保锁被释放的时间尽可能短，以避免 CPU 资源的浪费。
- 在多 CPU 环境下，自旋锁可能不如其他锁机制高效，因为它可能导致线程在不同的 CPU 上自旋等待。

结论

自旋锁是一种适用于短时间内锁竞争情况的同步机制，它通过减少线程切换的开销来提高锁操作的效率。然而，它也存在 CPU 资源浪费和可能引起活锁等缺点。在使用自旋锁时，需要根据具体的应用场景进行选择，并确保锁被释放的时间尽可能短。

样例代码

```C++
// 操作共享变量会有问题的售票系统代码
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

int ticket = 1000;
//pthread_spinlock_t lock;

void *route(void *arg)
{
    char *id = (char *)arg;
    while (1)
    {
        //pthread_spin_lock(&lock);
        if (ticket > 0)
        {
            usleep(1000);
            printf("%s sells ticket:%d\n", id, ticket);
            ticket--;
            //pthread_spin_unlock(&lock);
        }
        else
        {
            //pthread_spin_unlock(&lock);
            break;
        }
    }
    return nullptr;
}

int main(void)
{
    //pthread_spin_init(&lock, PTHREAD_PROCESS_PRIVATE);
    pthread_t t1, t2, t3, t4;

    pthread_create(&t1, NULL, route, (void *)"thread 1");
    pthread_create(&t2, NULL, route, (void *)"thread 2");
    pthread_create(&t3, NULL, route, (void *)"thread 3");
```

```
    pthread_create(&t4, NULL, route, (void *)"thread 4");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    //pthread_spin_destroy(&lock);
}
```