第 3 章　同步机制

一、源码阅读

阅读要求： 锁部分 spinlock.h spinlock.c 以及相关其他文件代码

源码阅读：

锁(Lock)是计算机操作系统中实现进程同步的重要机制。同步(Synchronization)是指让多个进程或线程能够按照程序员期望的顺序来协调执行。当多个进程或线程并发地执行并访问同一资源，并且进程/线程的执行结果依赖于其执行顺序，我们就称这种情况为竞争状态(Race Condition)。

Xv6 中实现了自旋锁(Spinlock)用于内核临界区访问的同步和互斥。自旋锁最大的特征是当进程拿不到锁时会进入无限循环，直到拿到锁退出循环。以下为自旋锁的数据结构，其定义在 spinlock.h 文件中。

```
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

其中，核心的变量 locked 的意义为：当 locked 为 1 时代表锁已被占用，反之未被占用，初始值为 0。

同时，在调用锁之前，必须对锁进行初始化。其初始化方法相对简单，即对锁名进行赋值，将 locked 变量与 cpu 变量赋值 0。

acquire()方法和 release()方法分别实现对 locked 变量进行原子操作占用锁和释放锁的功能。

acquire()方法：首先，禁止了中断，并且使用专门的 pushcli()方法，这个方法保证了中断的禁止；然后，采用 xchg 指令（内联汇编）来实现在设置 locked 为 1；同时，获得

其原来的值的操作；最后，使用__sync_synchronize 方法是为了避免编译器对这段代码进行指令顺序调整和 CPU 在这块代码采用乱序执行的优化。

```
void
acquire(struct spinlock *lk)
{
  pushcli();      // 关中断
  if(holding(lk)) // 如果能获得Lock，则返回
    panic("acquire");

  // xchg -> 交换两个变量
  // return &lk->locked
  // and lk->locked = 1
  // 自带锁总线机制
  // 非阻塞
  while(xchg(&lk->locked, 1) != 0)
    ;

  lk->cpu = cpu;    // 记录上锁的CPU
  getcallerpcs(&lk, lk->pcs);
}
```

release()方法：首先，为了保证设置 locked 为 0 的操作的原子性，同样使用了内联汇编；最后，使用 popcli()来允许中断。

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk)) // 如果 已经释放了
    panic("release");

  lk->pcs[0] = 0; // 释放栈指针
  lk->cpu = 0;

  xchg(&lk->locked, 0); // 同样的xchg 保证了写后读 避免OOD

  popcli();
}
```

同时，为了实现 acquire()方法和 release()方法的开关中断行为，spinlock.c 文件中还实现了 pushcli()方法和 popcli()方法

二、讨论总结

临界区(Critical Section)

临界区是指对共享数据进行访问与操作的代码区域。所谓共享数据，就是可能有多个代

码执行流并发地执行，并在执行中可能会同时访问的数据。

同步(Synchronization)与互斥(Mutual Exclusion)

同步是指让两个或多个进程/线程能够按照程序员期望的方式来协调执行的顺序。比如，让 A 进程必须完成某个操作后，B 进程才能执行。

互斥则是指让多个线程不能够同时访问某些数据，必须要一个进程访问完后，另一个进程才能访问。

竞争状态(Race Condition)

当多个进程/线程并发地执行并且访问一块数据，并且进程/线程的执行结果依赖于它们的执行顺序，我们就称这种情况为竞争状态。

临界区操作时中断是否开启？若中断开启，有何影响？

Xv6 操作系统要求在内核临界区操作时中断必须关闭。如果此时中断开启，那么可能会出现以下死锁情况：A 进程在内核态运行并拿下了 p 锁时，触发中断进入中断处理程序，中断处理程序也在内核态中请求 p 锁，由于锁在 A 进程手里，且只有 A 进程执行时才能释放 p 锁，因此中断处理程序必须返回，p 锁才能被释放。那么此时中断处理程序会永远拿不到锁，陷入无限循环，进入死锁。

XV6 的锁是如何实现的，有什么操作？

Xv6 中实现了自旋锁(Spinlock)用于内核临界区访问的同步和互斥。自旋锁最大的特征是当进程拿不到锁时会进入无限循环，直到拿到锁退出循环。具体操作，可参考上一节中源码阅读部分

xchg 指令是什么，该指令有何特性？

交换指令 XCHG 是两个寄存器，寄存器和内存变量之间内容的交换指令，两个操作数的数据类型要相同，可以是一个字节，也可以是一个字，也可以是双字。该指令的功能和 MOV 指令不同，后者是一个操作数的内容被修改，而前者是两个操作数都会发生改变。寄存器不能是段寄存器，两个操作数也不能同时为内存变量。 XCHG 指令不影响标志位。

三、方案设计

设计要求： 基于 XV6 的 spinlock，请给出实现信号量、读写锁、信号机制的设计方案

这里主要设计了读写锁，对信号量以及信号机制也有讨论和实现。

- **信号量**

基本方案：用 Xv6 提供的接口实现了信号量，格式和命名与 POSIX 标准类似。这个信号量的实现采用等待队列的方式。当一个进程因信号量陷入阻塞时，会将自己放进等待队列并睡眠。当一个进程释放信号量时，会从等待队列中取出一个进程继续执行。

伪码实现：

```
struct semaphore {
  int value;
  struct spinlock lock;    // 自旋锁
  struct proc *queue[NPROC];  // 等待队列
  int end;
  int start;
};

void sem_init(struct semaphore *s, int value) {
  s->value = value;
  initlock(&s->lock, "semaphore_lock");
  end = start = 0;
}

void P(struct semaphore *s) {
  acquire(&s->lock);  // 获取锁

  s->value--;
  if (s->value < 0) {
    s->queue[s->end] = myproc();  // 将进程加入到等待队列
    s->end = (s->end + 1) % NPROC;
    sleep(myproc(), &s->lock) // 睡眠
  }

  release(&s->lock);  // 释放锁
}

void V(struct semaphore *s) {
  acquire(&s->lock);  // 获取锁

  s->value++;
  if (s->value <= 0) {
    wakeup(s->queue[s->start]); // 唤醒等待队列队首进程
    s->queue[s->start] = 0;
    s->start = (s->start + 1) % NPROC;
  }

  release(&s->lock);  // 释放锁
}
```

- **读写锁**

基本方案： 由于 XV6 操作系统支持多处理器执行命令，因而最初方案本小组参考《多核计

算与程序设计》中对读写锁的设计：

1. 将读写操作定义为互斥关系

2. 将不同写操作定义为同步关系，并设置了读者计数器。

其数据结构为：

```
typedef struct RWLOCK_st{
    spinlock ReadLock; // 读锁
    spinlock WriteLock; // 写锁
    UINT uReadcount; // 读者计数器
}RWLOCK;
```

以下方法为对读锁与写锁的获取和释放方法。

```
RWLock_LockRead() {
ReadLock->acquire(); // 上锁锁住计数器变量的读写，即读锁
uReadcount += 1; // 读者计数器加1
if(uReadcount > 0) // 判断是否触发写锁获取条件——读者非零
WriteLock->acquire(); // 获取写锁
ReadLock->release(); // 解锁计数器变量的读写，即读锁
}
RWLock_UnlockRead() {
ReadLock->acquire(); // 上锁锁住计数器变量的读写，即读锁
uReadcount -= 1; // 读者计数器减1
if(uReadcount == 0) // 判断是否触发写锁释放条件——读者为零
WriteLock->release(); // 释放写锁
ReadLock->release(); // 解锁计数器变量的读写，即读锁
}

RWLock_LockWrite() {
WriteLock->acquire(); // 获取写锁
}
RWLock_UnlockWrite() {
WriteLock->release(); // 释放写锁
}
```

优点：保证了多处理器上的读写操作的同步互斥关系 实现简单。

缺点：读操作比较频繁时，计数 uReadCount 可能一直无法归零，会导致写操作饿死现象 ，消耗较大

优化方案： 为读操作设置副本文件，使得读写操作不存在互斥关系。

RWLock_LockRead()：

1. 在获取读锁后，立即判断读者计数器是否为 0。
    1.1. 若为 0，则建立副本文件并链接至读写锁（副本文件唯一）；
2. 读者计数器加 1

3. 释放读锁

RWLock_UnlockRead()：

1. 获取读锁
2. 读者计数器减 1
3. 判断读者计数器是否为 0，若为 0，则取消链接并删除副本文件；
4. 释放读锁

优点：解决了写操作可能出现的饿死现象

缺点：副本文件与原文件可能存在滞后的问题。

- **信号机制**

基本方案：条件变量是管程内的等待机制。进入管程的线程因资源被占用而进入等待状态

每一个条件变量均表示一种等待原因，并且对应一个等待队列。其数据结构为：

```
Class Condition{
    int wait_count = 0; // 等待的线程数目
    Wait_Queue wait_queue; // 等待队列
}
```

Wait()方法和 Signal()方法的伪码实现

```
Condition::Wait(lock){
    wait_count++;
    Add this thread t to wait_queue;          // 将当前线程添加到等待队列
    release(lock);                            // 释放锁
    schedule();                               // 需要保证原子性与一致性
    require(lock);                            // 申请锁，实现互等
}

Condition::Signal(){
    if(wait_count>0){                         // 如果等待的线程数目大于0
        Remove a thread t from wait_queue;    // 从等待队列移除一个就绪线程
        wakeup(t);                            // 需要保证原子性与一致性
        wait_count--;
    }
}
```