Lab3同步机制实验报告

Exercise 1

中断机制

- 禁用中断：linux内核提供了两个宏：`local_irq_enable`，`local_irq_disable`，分别用来申请开关中断，使得进程可以实现同步功能

锁机制

- 自旋锁：linux提供了自旋锁，定义在头文件`src/include/linux/spinlock_types.h`中，其结构体如下。

```
typedef struct spinlock {
        union {
                struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
                struct {
                        u8 __padding[LOCK_PADSIZE];
                        struct lockdep_map dep_map;
                };
#endif
        };
} spinlock_t;
```

- 读写锁：读写锁的存在，是针对特定的数据读写情况设计的特殊自旋锁，能够并发进行读操作，进而提高程序和系统的性能，其结构体定义在`src/include/linux/rwlock_types.h`中，结构体如下：

```
typedef struct {
        arch_rwlock_t raw_lock;
#ifdef CONFIG_DEBUG_SPINLOCK
        unsigned int magic, owner_cpu;
        void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map dep_map;
#endif
} rwlock_t;
```

- 顺序自旋锁,上述的读写锁存在写进程过长等待的情况，因此，顺序自旋锁seqlock给写进程提供更高的优先级，写进程可以抢占读进程，如果读进程被抢占，则此次读取无效，其结构体定义在`src/include/linux/seqlock.h`中，其结构体维护了变量`sequence`，每次写操作都需要更新该变量，而读操作则需要在读取开始和结束检查变量是否变化来确认自己是否曾经被抢占，因此，其官方推荐操作方式如下：

```
* Expected non-blocking reader usage:
*      do {
*          seq = read_seqbegin(&foo);
*      ...
*      } while (read_seqretry(&foo, seq));
```

信号量机制

- 普通信号量：普通信号量除了初始化，只提供`UP，DOWN`两种操作，分别对应释放和申请临界区的控制权，其定义在`src/include/linux/semaphnore.h`中,其结构体如下：

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

- 读写信号量，其作用类似于读写锁，适用于读写数据的情况。
- RCU（Read-Copy-Update）操作，也是针对读写操作的一种同步机制，其操作原则应该遵循如下规则
    - 对于读者：调用`rcu_read_lock`和`rcu_read_unlock`函数构建读者临界区，其内部直接获得共享数据的内存指针，能够对其进行读操作，但是

操作应该在临界区内部完成
    - 对于写者：首先需要分配新的空间写入数据，然后更新原来的指针为新的指针，达到写的目的

Exercise 3

锁机制的实现

基于信号量实现原子操作，其结构体和相关方法定义如下：

```
class Lock
{
public:
    Lock(char *debugName);           // initialize lock to be FREE
    ~Lock();                         // deallocate lock
    char *getName() { return name; } // debugging assist

    void Acquire(); // these are the only operations on a lock
    void Release(); // they are both *atomic*

    bool isHeldByCurrentThread(); // true if the current thread
                                  // holds this lock.  Useful for
                                  // checking in Release, and in
                                  // Condition variable ops below.

private:
    char *name; // for debugging
                // plus some other stuff you'll need to define

    Thread* heldThread;
    Semaphore* semaphore;
};

Lock::Lock(char *debugName)
{
    this->semaphore = new Semaphore(debugName, 1);
    this->heldThread = NULL;
    this->name = debugName;
}
Lock::~Lock()
{
    delete this->semaphore;
}
```

```
bool Lock::isHeldByCurrentThread()
{
    return heldThread == currentThread;
}
/*
    获取锁，如果当前锁busy则sleep,获取成功则设置当前的持有线程
*/
void Lock::Acquire()
{
    this->semaphore->P();
    this->heldThread = currentThread;
}
/*
    释放锁，如果当前线程不是持有锁的线程，则直接反回，无权释放，
*/
void Lock::Release()
{
    if (!isHeldByCurrentThread())
    {
        return;
    }
    this->semaphore->V();
    heldThread = NULL;
}
```

条件变量实现

实现代码如下：

```
class Condition
{
public:
    Condition(char *debugName); // initialize condition to
                                // "no one waiting"
    ~Condition();               // deallocate the condition
    char *getName() { return (name); }

    void Wait(Lock *conditionLock);      // these are the 3
operations on
                                         // condition variables;
releasing the
                                         // lock and going to sleep
are
```

```
                                        // *atomic* in Wait()
    void Signal(Lock *conditionLock);   // conditionLock must be
held by
    void Broadcast(Lock *conditionLock); // the currentThread for
all of
                                        // these operations

private:
    char *name;
    // plus some other stuff you'll need to define
    List* queue;
};
Condition::Condition(char *debugName)
{
    this->name = debugName;
    this->queue = new List;
}
Condition::~Condition()
{
    delete queue;
}
/*
    等待在条件变量上，操作：加入队列， 释放锁，调度，申请锁
*/
void Condition::Wait(Lock *conditionLock)
{
    this->queue->Append(currentThread);
    conditionLock->Release();
    currentThread->Sleep();
    conditionLock->Acquire();
}
void Condition::Signal(Lock *conditionLock)
{
    if (!this->queue->IsEmpty())
    {
        Thread *t = (Thread *)this->queue->Remove();
        if (t != NULL)
        {
            scheduler->ReadyToRun(t);
        }
    }
}
void Condition::Broadcast(Lock *conditionLock)
```

```
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    while (!this->queue->IsEmpty())
    {
        Thread *t = (Thread *)this->queue->Remove();
        if (t != NULL)
        {
            scheduler->ReadyToRun(t);
        }
    }
    (void)interrupt->SetLevel(oldLevel);
}
```

Exercise 4

基于锁机制的读者写者问题：

```
/*
    读者写者问题：基于锁机制
*/
class ReadWriteLock{
    Lock* writeLock;   //控制写操作
    Lock* readerNumLock;  //控制读者数量
    int readerNum;   //读者数量

    /*  读者在读取数据时需要检查是否有写者在写，表现为可写的锁busy ，
        第一个读者检查可写锁，其余读者则检查读者数量
    */
   ReadWriteLock(){
       writeLock = new Lock("write lock");
       readerNumLock = new Lock("reader num lock");
       readerNum = 0;
   }
   void read();
   void write();
    void reader(){
        readerNumLock->Acquire();
        readerNum ++;  //增加读者数量
        if(readerNum == 1){   //第一个读者，锁住写操作
            writeLock->Acquire();
        }
        readerNumLock->Release();
```

```
        read();   //读取数据

        readerNumLock->Acquire();
        readerNum --;
        if(readerNum == 0){   // 这是最后一个读者
            writeLock->Release();
        }
        readerNumLock->Release();
    };
    void writer(){
        writeLock->Acquire();
        write();
        writeLock->Release();
    };
};
```

基于条件变量的读者写者问题

```
/*  读者写者问题，基于条件变量机制  */
class ReadWriteCondition{
    int AR; //在读者数量
    int WR; //等待读者数量
    int AW; //写者数量
    int WW; //等待读者数量
    Lock* lock;
    Condition* toRead;   //读者等待队列
    Condition* toWrite;  //写者等待队列

    void read();
    void write();
    /*
        读者检查是否有读者在读，如果有人在读，则直接读，如果没人读，则检查是否有人写，
        如果有人写，则在读者队列休眠,否则作为第一个读者给写锁
    */
    void reader(){
       lock->Acquire();
       while(AW + WW > 0){  //有读者在写或者在等
            WR ++;
            toRead->Wait(lock);   //等待
            WR --;
       }
       AR ++;  //读者数量增加
```

```
        lock->Release();

        read();

        lock->Acquire();
        AR --;
        if(AR == 0 && WW > 0){
            toWrite->Signal(lock);
        }
         lock->Release();
    };
    /*
        如果有人在写或者有人在读，都需要休眠，否则则可以写，写完成后选择唤醒写进程或者读进程
    */
    void writer(){
        lock->Acquire();
        while(AR + AW > 0){
            WW ++;
            toWrite->Wait(lock);
            WW --;
        }
        AW ++;
        lock->Release();

        write();

        lock->Acquire();
        AW --;
        if(WW > 0){
            toWrite->Signal(lock);
        }else if(WR > 0){
            toRead->Signal(lock);
        }
        lock->Release();
    };
};
```

Challenge 1

实现多线程到达指定位置的同步操作，设计如下结构，其中记录已经达到的线程数量和需要到达的线程数量，每个线程到达指定位置后判断是否所有进程都已到达，如果否，则休眠在指定信号量队列上，否则则唤醒队列中所有的进程，代码如下：

```
/*  进程同步，多个进程同时到达指定位置可以继续执行  */
class Barrier{
    Lock* lock;
    int waitingNum;  // 已有几个线程到达指定位置
    int runNum;  //需要几个线程到达指定位置
    Condition* toRun;

    void run(){
        /*
        ... 执行各个进程相关操作，到达指定位置
        */
        lock->Acquire();
        if(waitingNum < runNum){
            waitingNum ++;
            toRun->Wait(lock);
        }else{
            toRun->Broadcast(lock);
        }
        /*
        ... 执行后续操作
        */
    }
};
```

Challenge 3

linux中的kfifo机制，其实现原理为维护一个环形的缓冲队列，分别设置读指针和写指针，读者读取时使用`out`变量指向的位置，而写者写入时使用`in`变量指向的位置，来保证其不会发生读写冲突，其适用范围为：单个读者单个写者的情况，如果有多个读者写者或者有多个cpu并行，则同样需要设置锁或者设置内存屏障，以达到同步的效果，对于Nachos来说，其只有单个CPU，因此针对单线程读者和单线程写者的情况，可以移植该机制到Nachos中。同时可能需要对Nachos中保存数据的相关结构进行更改，设定32位数据均为无符号数，因为kfifo机制并不对数字取模，而是使其自行溢出（当然移植的时候可以变为取模操作）