Chapter 7 Homework

陈文迪 519021910071

作业中的引用内容均已标出

7.8 Linux 内核采取了如下策略：一个进程不能在尝试获取一个信号量的同时持有一个自旋锁。请解释为什么要制定这样的策略。

问题解答：

> 当一个进程执行操作wait（）并且发现信号量值不为正时，它必须等待。然而，该进程不是忙等待而是阻塞自己。阻塞操作将一个进程放到与信号量相关的等待队列中，并且将该进程状态切换成等待状态。

这就导致该进程会持续等待状态一段时间，此时若其还同时持有一个自旋锁，那么在此期间其他与该自旋锁相关的进程会陷入忙等待。等待状态的时间一般较长，那么这种情况会导致CPU周期的大量浪费，降低系统的整体效率。

7.11 讨论在读者-作者问题中公平性和吞吐量的权衡。试着提出一种方法在解决读者-作者问题的同时也不会造成饥饿。

问题解答：

在读者-作者问题中，通过允许多个读进程同时阅读来增大吞吐量。但这样也减少了公平性，因为这样的解决方案更偏向于读者，作者有可能无限等待读进程的完成，造成饥饿。

为了解决饥饿问题，我们事实上需要维护进程到达的顺序。通过阅读信号量的实现我们可以发现，当一个进程等待信号量时，它会被挂起到一个等待队列中，当有进程调用 `signal()` 后会有一个进程被从等待进程链表上取下，这实际上就实现了FIFO的队列（Linux系统文档中并没有明确，但实际实现上确实是FIFO的，或者我们可以自己设计数据结构来实现）。我们不妨将这个信号量初始化为1，并记作 `order_mutex`。在此基础上，我们可以给出修改后的读者和作者代码。

对于作者：

```
do{
    wait(order_mutex);
    wait(rw_mutex);
    signal(order_mutex);

    /* writing */

    signal(rw_mutex);
}while(true);
```

对于读者：

```
do{
    wait(order_mutex);
    wait(mutex);
    read_count++;
    if(read_count==1)
        wait(rw_mutex);
    signal(order_mutex);
```

```
    signal(mutex);

    /* reading */

    wait(mutex);
    read_count--;
    if(read_count==0)
        signal(rw_mutex);
    signal(mutex);
}while(true);
```

7.16 请分析如下的同步问题。

在 `stack-ptr.c` 文件中使用链表实现了一个栈。它的用法示例如下：

```
StackNode *top = NULL;
push(5, &top);
push(10, &top);
push(15, &top);

int value = pop(&top);
value = pop(&top);
value = pop(&top);
```

该程序目前存在一个竞争条件并且不适用于一个并发系统。请使用Pthread中的互斥锁来解决这个竞争条件。

问题解答：

可以看到，对于这样的操作原语，共享变量在于 `top` 指针，而关键区从调用 `push(x, &top)` 和 `pop(&top)` 时就已经开始了。因此我们可以对 `push(x, &top)` 和 `pop(&top)` 做如下替换。利用互斥锁来保证同步。

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);

pthread_mutex_lock(&mutex);
push(x, &top);
pthread_mutex_unlock(&mutex);

pthread_mutex_lock(&mutex);
pop(&top);
pthread_mutex_unlock(&mutex);
```

当然，我们可以用面向对象的方法对其做进一步封装。

```
class stack{
private:
    struct node{
        ...
    }
    pthread_mutex_t mutex;
    node *top;

public:
    stack(){
```

```
        pthread_mutex_init(&mutex,NULL);
        top = NULL;
    }
    void push(int x){
        pthread_mutex_lock(&mutex);
        ...
        pthread_mutex_unlock(&mutex);
    }
    int pop(){
        pthread_mutex_lock(&mutex);
        ...
        pthread_mutex_unlock(&mutex);
    }
}
```