Week13 同步互斥

一、 实验概述

在前几次的实验中我们已经实现了进程以及调度算法，可以让多个进程并发的执行。在现实的系统当中，多线程的系统都需要协同的完成某一项任务。但是在协同的过程中，存在许多资源共享的问题，比如对一个文件读写的并发访问等等。这些问题需要我们提供一些同步互斥的机制来让程序可以有序的、无冲突的完成他们的工作。我们本次实验将在ucore中解决这些问题。

二、实验目的

1. 理解操作系统的同步互斥的设计实现
2. 掌握在ucore中信号量机制的具体实现
3. 理解管程机制，在ucore中增加基于管程的条件变量的支持
4. 了解经典进程同步问题，并能使用同步机制解决进程同步问题

三、实验项目整体框架概述

// Week13
├── kern
│ ├── debug
│ ├── driver
│ ├── fs
│ ├── init
│ ├── libs
│ ├── mm
│ ├── process
│ ├── schedule
│ ├── **sync**
│ │ ├── **check_sync.c** //哲学家算法
│ │ ├── **sem.c** //信号量
│ │ ├── **sem.h**
│ │ ├── sync.h
│ │ ├── **wait.c** //等待队列
│ │ └── **wait.h**
│ ├── syscall
│ └── trap
├── libs
├── Makefile
├── tools
└── user

四、实验内容

为了提供同步互斥机制，操作系统有多种实现方法，本次实验我们将介绍以下实现方式：

1. 屏蔽使能中断（已经用到）

这部分主要是处理内核内的同步互斥问题。因为内核在执行的过程中可能会被外部的中断打断，我们实现的ucore是不可抢占的系统，所以操作系统进行某些同步互斥的操作的时候需要先禁用中断，等执行完之后再打开中断。这样保证了操作系统在执行临界区的时候不会被打断，也就实现了同步互斥。trap包括中断和异常，异常是无法屏蔽的。

在ucore中经常可以看到下面这样的代码：

```
......
local_intr_save(intr_flag);
{
  临界区代码
}
local_intr_restore(intr_flag);
......
```

这段代码就是使用屏蔽使能中断处理内核同步互斥问题的例子。

2. 信号量

信号量（semaphore）是一种同步互斥的实现。semaphore一词来源于荷兰语，原来是指火车信号灯。想象一些火车要进站，火车站里有n个站台，那么同一时间只能有n辆火车进站装卸货物。当火车站里已经有了n辆火车，信号灯应该通知后面的火车不能进站了。当有火车出站之后，信号灯应该告诉后面的火车可以进站。

这个问题放在操作系统的语境下就是有一个共享资源只能支持n个线程并行的访问，信号量统计目前有多少进程正在访问，当同时访问的进程数小于n时就可以让新的线程进入，当同时访问的进程数为n时想要访问的进程就需要等待。

上节课我们已经了解并使用过posix中的信号量，这节课我们需要实现ucore中的信号量。

在信号量中，一般用两种操作来刻画申请资源和释放资源：P操作申请一份资源，如果申请不到则等待；V操作释放一份资源，如果此时有进程正在等待，则唤醒该进程。在ucore中，我们使用`down`函数实现P操作，`up`函数实现V操作。

结构体

首先是信号量结构体的定义：

```
typedef struct {
    int value;
    wait_queue_t wait_queue;
} semaphore_t;
```

其中的`value`表示信号量的值，其正值表示当前可用的资源数量，负值表示正在等待资源的进程数量。`wait_queue`即为这个信号量相对应的等待队列。

等待队列

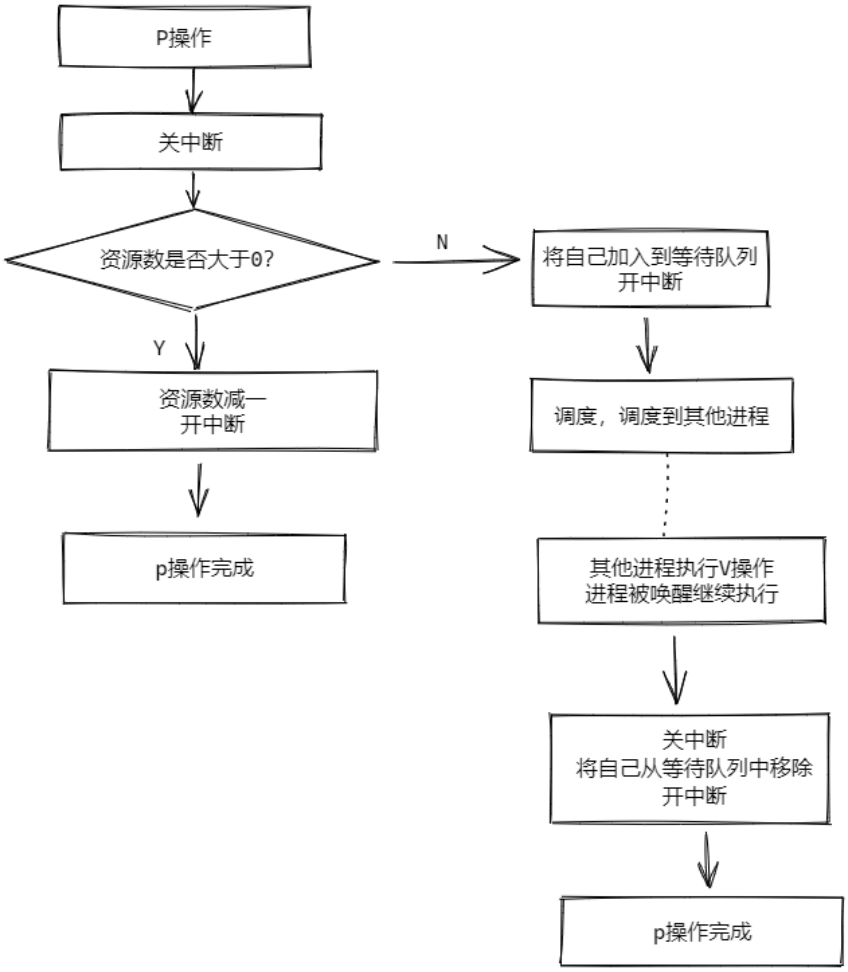
等待队列是操作系统提供的一种事件机制。一些进程可能会在执行的过程中等待某些特定事件的发生，这个时候进程进入睡眠状态。操作系统维护一个等待队列，把这个进程放进他等待的事件的等待队列中。当对应的事件发生之后，操作系统就唤醒相应等待队列中的进程。

等待队列的实现用到了前面实验中经常用到的双向链表，主要数据结构和方法是定义在`kernel\sync\wait.c`中。请大家自行阅读此部分代码。

P操作

`down`函数对应的是P操作。首先关闭中断，然后判断信号量的值是否为正。如果是正值说明进程可以获得信号量，将信号量的值减一，打开中断然后函数返回即可。否则表示无法获取信号量，将自己的进程保存进等待队列，打开中断，调用`schedule`函数进行调度。

等到V操作唤醒等待队列中的进程的时候，进程会从`schedule`函数后面开始执行，这时候将自身从等待队列中删除（此过程需要关闭中断）并返回即可。



具体的实现如下：

```
static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    if (sem->value > 0) {
        sem->value --;
        local_intr_restore(intr_flag);
        return 0;
    }
    wait_t __wait, *wait = &__wait;
    wait_current_set(&(sem->wait_queue), wait, wait_state);
    local_intr_restore(intr_flag);

    schedule();

    local_intr_save(intr_flag);
    wait_current_del(&(sem->wait_queue), wait);
    local_intr_restore(intr_flag);
```

```
    if (wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;
    }
    return 0;
}
```

V操作

`up`函数实现了V操作。首先关闭中断，如果释放的信号量没有进程正在等待，那么将信号量的值加一，打开中断直接返回即可。如果有进程正在等待，那么唤醒这个进程，把它放进就绪队列，打开中断并返回。具体的实现如下：

```
static __noinline void __up(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        wait_t *wait;
        if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) {
            sem->value ++;
        }
        else {
            assert(wait->proc->wait_state == wait_state);
            wakeup_wait(&(sem->wait_queue), wait, wait_state, 1);
        }
    }
    local_intr_restore(intr_flag);
}
```

六、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 如何屏蔽中断
2. 为何屏蔽中断
3. 如何实现信号量

七、下一实验简单介绍

下一次实验，我们将介绍ucore中实现文件系统。