



## “操作系统原理与实践”实验报告

### 信号量的实现和应用

本实验所有代码在这里：

[http://git.shiyanlou.com/gaoyuanhezhihao/shiyanlou\\_cs115/src/master/Lab6](http://git.shiyanlou.com/gaoyuanhezhihao/shiyanlou_cs115/src/master/Lab6)

首先编写信号量测试用户代码：

#### 实验数据

学习时间 1593分钟

操作时间 966分钟

按键次数 0次

实验次数 9次

报告字数 10386字

是否完成 完成

#### 评分

# 未评分

下一篇

篇

#### 相关报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 基于内核栈切换的进程切换 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 信号量的实现和应用 实验报告

```

#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <fcntl.h>
#define FILECELLSIZE 3
FILE *fpWork;
FILE *fpFakeTerminal;
int main(int argc, char **argv)
{

    sem_t sem_empty, *pSem_empty=&sem_empty;
    sem_t sem_full, *pSem_full=&sem_full;
    sem_t sem_mutex, *pSem_mutex=&sem_mutex; //only one process can handle the
file at the same time
    printf("main start");
    fflush(stdout);
    int iwait=0;
    int WriteReadIndex[2]={0,0};
    //系统中可能已经存在着三个信号量，把它们清楚掉
    sem_unlink("empty");
    sem_unlink("full");
    sem_unlink("mutex");
    //建立这三个型号量
    pSem_empty=sem_open("empty",O_CREAT | O_EXCL,0600,10);
    pSem_full=sem_open("full",O_CREAT | O_EXCL,0600,0);
    pSem_mutex=sem_open("mutex",O_CREAT | O_EXCL,0600,1);
    /*打开作为缓存的文件。文件的书写格式是：写进程从文件的4号字节开始顺序写入内容，读进程
也从4号字节开始顺序读进程内容。为了不同的写进程之间同步写的位置，文件的第0，1字节用来
保存当前写的位置。同理，为了读进程之间能够同步文件的第2，3用来保存读的位置（这里也可以
使用一个信号量来同步读写位置）。为了简化，这里并没有完全按照实验指导中的要求，把读过的
内容删掉。*/
    fpWork = fopen("work.dat", "wb+");
    fwrite(WriteReadIndex, 2*sizeof(int), 1, fpWork);
    fflush(fpWork);
    if(!fork())
    {
        //this is subprocess of writing data
        //写进程
        printf("writer process start\n");
        fflush(stdout);
        int i=0;
        while(i <= 50)
        {
            int FileWriteIndex = 0;
            ++i; //produce a item
            sem_wait(pSem_empty); //P(Empty)
            sem_wait(pSem_mutex); //P(Mutex)

            // #1读取文件中记录的写序号（接下来应该写的位置）
            fseek(fpWork, 0*sizeof(int), SEEK_SET); //把文件指针指向文件第一个字节
            fread(&FileWriteIndex, sizeof(int), 1, fpWork);
            // #2写入
            fseek(fpWork, (FileWriteIndex+2)*sizeof(int), SEEK_SET);
            fwrite(&i, sizeof(int), 1, fpWork);
            fflush(fpWork);
            printf("Write Index:%d,%d is writed\n", FileWriteIndex, i);
            fflush(stdout);
            // #3更新写序号
            fseek(fpWork, 0*sizeof(int), SEEK_SET);
            FileWriteIndex++;
            fwrite(&FileWriteIndex, sizeof(int), 1, fpWork);
            fflush(fpWork);

            sem_post(pSem_mutex); //V(Mutex)
            sem_post(pSem_full); //V(Full)
        }
        while(1);
    }
    if(!fork())
    {
        //this is the first subprocess of reading data
        //第一个写进程
        int FileReadIndex_1 = 0;
        while(1)
        {
            int a=0;
            sem_wait(pSem_full);
            sem_wait(pSem_mutex);
            // #1
            fseek(fpWork, 1*sizeof(int), SEEK_SET);

```

```

        fread(&FileReadIndex_1, sizeof(int), 1, fpWork);
        // #2
        fseek(fpWork, (FileReadIndex_1+2)*sizeof(int), SEEK_SET);
        fread(&a, sizeof(int), 1, fpWork);
        printf("ReadIndex1:%d,%d:%d\n", FileReadIndex_1, getpid(), a);
        // #3
        fseek(fpWork, 1*sizeof(int), SEEK_SET);
        FileReadIndex_1++;
        fwrite(&FileReadIndex_1, sizeof(int), 1, fpWork);
        fflush(stdout);
        fflush(fpWork);
        sem_post(pSem_mutex);
        sem_post(pSem_empty);
    }
}
if(!fork())
{
    //this is the second subprocess of reading data
    //第二个写进程
    int FileReadIndex_2 = 0;
    while(1)
    {
        int j=0;
        sem_wait(pSem_full);
        sem_wait(pSem_mutex);
        // #1
        fseek(fpWork, 1*sizeof(int), SEEK_SET);
        fread(&FileReadIndex_2, sizeof(int), 1, fpWork);
        // #2
        fseek(fpWork, (FileReadIndex_2+2)*sizeof(int), SEEK_SET);
        fread(&j, sizeof(int), 1, fpWork);
        printf("ReadIndex2:%d,%d:%d\n", FileReadIndex_2, getpid(), j);
        // #3
        fseek(fpWork, 1*sizeof(int), SEEK_SET);
        FileReadIndex_2++;
        fwrite(&FileReadIndex_2, sizeof(int), 1, fpWork);
        fflush(fpWork);
        fflush(stdout);
        sem_post(pSem_mutex);
        sem_post(pSem_empty);
    }
}
wait(&iwait);
}

```

在ubuntu下运行正常，输出如第一幅截图所示。

向linux-0.11添加


```

sem_t sem_open(const char *name, unsigned int value);
int sem_wait(sem_t sem);
int sem_post(sem_t sem);
int sem_unlink(const char *name);

```

四个系统调用，为了简化，并没有使用sem\_t \*作为信号量标记，而是直接用sem\_t（#define为int）变量作为信号量唯一的标识。

1.在include/unistd.h中添加四个系统调用功能号

 图片描述 注意，因为系统调用号增加了，而在int 0x80中断处理函数sys\_call函数中会判断中断是否在有效的范围内，所以这里还需要修改system\_call.s中的nr\_system\_calls

2.在include/linux/sys.h的sys\_call\_table中添加这四个函数指针

3.实现这四个系统调用。这一步是实验的关键，也是最难的地方。主要参考的是老师18讲的内容。首先定义信号量等待任务结构：

```

struct Sem_wait_list{
    task_struct *TaskNode;
    char name[20];
    int value;
};

```

一个这样的对象表示一个信号量，信号量的名字在name中保存。同时TaskNode成员指向的是等待消费这个信号的任务。value部分表示的是信号值。为了简化，我们设置系统中最多的信号量个数是80。所有的信号量保存在信号数组中，如下：

```

struct Sem_wait_list SemArray[80]={0}; //Max number of semaphony is 80

```

信号量的表示sem\_t定义为int变量，这个变量也表示信号量在信号数组中的索引。升级版程序中可以考虑使用链表数据结构来代替这里的数组。

sys\_sem\_open 函数首先查找当前信号数组，看是否已经存在这一信号量，如果存在，则返回sem\_t（在这里也就是数组下标）。如果不存在，则新建一个，再返回sem\_t。程序中需要注意的是，在sem\_open函数中，通过一个const char \*name 参数传递信号量的名字。但是指针参数传递的是应用程序所在地址空间的逻辑地址，在内核中如果直接访问这个地址，访问到的是内核空间中的数据，不是用户空间的。这一点可以参考前面系统调用的实验指导。

```
sem_t sys_sem_open(const char *name,unsigned int value)
{
    int i=0;
    int LastEmpty = 0;
    char UserMemoryByte[2]={'\0','\0'};
    UserMemoryByte[0] = get_fs_byte(name);
    printk("sys_sem_open:name:%s,value:%d\n",UserMemoryByte ,value);
    for(i=1;i<MAX_SEMARRAY_NUM;i++)
    {
        if(SemArray[i].name[0] =='\0')
        {
            LastEmpty = i;
            continue;
        }
        if(!strcmp(SemArray[i].name,UserMemoryByte))
        {
            printk("sys_sem_open:find the sem\n");
            return i;
        }
    }
    //Not found
    if(0 == LastEmpty)
    {
        //The array is full
        return -1;
    }
    //Add new semaphony
    printk("sys_sem_open:add new sem\n");
    strcpy(SemArray[LastEmpty].name ,UserMemoryByte);
    SemArray[LastEmpty].value = value;
    return LastEmpty;
}
```

为了简化程序的编写难度，在这个函数中，我只使用了传入名称字符串的第一个字符作为 name 标识：

```
char UserMemoryByte[2]={'\0','\0'};
UserMemoryByte[0] = get_fs_byte(name);
```

这个方法只是一时求快之举，实际上这样是很危险的。“empty”, "elementary"将会被识别为同一个信号量。

sys\_sem\_wait函数中首先查找这个信号量是否存在，如果不存在则返回错误。如果存在则判断这一信号量是否已小于或等于0，如果是则进入睡眠。否则把信号量减一，继续前进。

```
int sys_sem_wait(sem_t sem)
{
    cli();
    if(sem >= MAX_SEMARRAY_NUM || sem<= 0 )
    {
        printk("sys_sem_wait:sem:%d is out of range\n",sem);
        sti();
        return -1;
    }
    if( SemArray[sem].name[0] == '\0' )
    {
        //This semophony does'n exist
        sti();
        return -1;
    }
    while( 0 >= SemArray[sem].value )
    {
        sleep_on( &(amp;SemArray[sem].TaskNode) );
    }

    //Catch the symphony
    --SemArray[sem].value;

    sti();
}
```

sys\_sem\_post函数主要是生产函数，在这个函数中首先也是判断参数所指的信号量是否存在，如果不存在则返回，如果不存在则返回。如果存在则把信号量的value加一。再唤醒等待队列中的任务。

```
int sys_sem_post(sem_t sem)
{
    cli();
    if( SemArray[sem].name[0] == '\0' )
    {
        //This semophony does'n exist
        sti();
        return -1;
    }
    ++SemArray[sem].value;
    wake_up(  &(SemArray[sem].TaskNode) );
    sti();
}
```

sys\_sem\_unlink函数相对比较简单。首先查找这一信号量，如果不存在则返回错误。如果存在则从信号量数组SemArray中删除对应的成员。删除的具体步骤就是把对应成员重置为0。

```
int sys_sem_unlink(const char *name)
{
    char UserMemoryByte[2]={'\0','\0'};
    UserMemoryByte[0] = get_fs_byte(name);
    int i=0;
    for(i=1;i<MAX_SEMARRAY_NUM;i++)
    {
        if( !strcmp(SemArray[i].name,UserMemoryByte) )
        {
            SemArray[i].name[0] = '\0';
            SemArray[i].TaskNode = 0;
            SemArray[i].value=0;
            return 0;
        }
    }
}
```

4.编写这四个系统调用对应的API。在include下新建sem.h文件。内容如下：

```
#define __LIBRARY__
#include <unistd.h>
_syscall2(int,sem_open,const char *,name,unsigned int,value)
_syscall1(int,sem_wait,int,sem)
_syscall1(int,sem_post,int,sem)
_syscall1(int,sem_unlink,const char *,name)
```

5.编写测试程序，在前面PC.c的基础上进行改写。这里特别提出的是，在使用printf函数输出之后，要使用fflush(stdout)把缓存中的内容“冲”出去。不大会导致一些莫名其妙的问题。鄙人因为忘记在输出后加这句，导致调试了3个小时。

```

#include <sem.h>
#include <stdio.h>
#define sem_t int
#define FILECELLSIZE 3
FILE *fpWork;
FILE *fpFakeTerminal;
int main(int argc, char **argv)
{
    sem_t sem_empty;
    sem_t sem_full;
    sem_t sem_mutex;
    int iwait=0;
    int WriteReadIndex[2]={0,0};
    int i = 0;
    int FileWriteIndex = 0;
    int FileReadIndex_1 = 0;
    int a=0;
    int FileReadIndex_2 = 0;
    int j=0;
    printf("main start");
    fflush(stdout);
    sem_unlink("empty");
    sem_unlink("full");
    sem_unlink("mutex");
    sem_empty=sem_open("empty",10);
    sem_full=sem_open("full",0);
    sem_mutex=sem_open("mutex",1);
    fpWork = fopen("work.dat","wb+");
    fwrite(WriteReadIndex,2*sizeof(int),1,fpWork);
    fflush(fpWork);
    printf("Opened file:%d\n",fpWork);
    fflush(stdout);
    if(!fork())
    {
        printf("writer process start\n");
        fflush(stdout);
        i=0;
        while(i <= 50)
        {
            ++i;
            sem_wait(sem_empty);
            sem_wait(sem_mutex);
            fseek(fpWork,0*sizeof(int),SEEK_SET);
            fread(&FileWriteIndex,sizeof(int),1,fpWork);
            fseek(fpWork,(FileWriteIndex+2)*sizeof(int),SEEK_SET);
            fwrite(&i,sizeof(int),1,fpWork);
            fflush(fpWork);
            printf("Write Index:%d,%d is writed\n",FileWriteIndex,i);
            fseek(fpWork,0*sizeof(int),SEEK_SET);
            FileWriteIndex++;
            fwrite(&FileWriteIndex,sizeof(int),1,fpWork);
            fflush(fpWork);
            sem_post(sem_mutex);
            sem_post(sem_full);
        }
        while(1);
    }
    if(!fork())
    {
        while(1)
        {
            sem_wait(sem_full);
            sem_wait(sem_mutex);

            fseek(fpWork,1*sizeof(int),SEEK_SET);
            fread(&FileReadIndex_1,sizeof(int),1,fpWork);
            fseek(fpWork,(FileReadIndex_1+2)*sizeof(int),SEEK_SET);
            fread(&a,sizeof(int),1,fpWork);
            printf("ReadIndex1:%d,%d:%d\n",FileReadIndex_1,getpid(),a);
            fseek(fpWork,1*sizeof(int),SEEK_SET);
            FileReadIndex_1++;
            fwrite(&FileReadIndex_1,sizeof(int),1,fpWork);
            fflush(stdout);
            fflush(fpWork);
            sem_post(sem_mutex);
            sem_post(sem_empty);
        }
    }
    if(!fork())
    {

```

```

while(1)
{
    j=0;
    sem_wait(sem_full);
    sem_wait(sem_mutex);
    fseek(fpWork,1*sizeof(int),SEEK_SET);
    fread(&FileReadIndex_2,sizeof(int),1,fpWork);
    fseek(fpWork,(FileReadIndex_2+2)*sizeof(int),SEEK_SET);
    fread(&j,sizeof(int),1,fpWork);
    printf("ReadIndex2:%d,%d:%d\n",FileReadIndex_2,getpid(),j);
    fseek(fpWork,1*sizeof(int),SEEK_SET);
    FileReadIndex_2++;
    fwrite(&FileReadIndex_2,sizeof(int),1,fpWork);
    fflush(fpWork);
    fflush(stdout);
    sem_post(sem_mutex);
    sem_post(sem_empty);
}
}
wait(&iwait);
}

```

6.调试过程的一些问题。因为在bochs运行linux0.11的时候，大量的输出会导致花屏，可以把内容导出到一个文件中。比如你的程序是a.out。那么使用 ./a.out >> Print.txt函数可以把输出写到Print.txt中。有时候通过ctrl+c停下程序后，屏幕一片空白，敲键盘输入也没有显示。此时只是显示有点问题。直接闭着眼睛敲命令，用vi 打开这个文件即可：vi Print.txt。鄙人在调试的过程中，在系统调用处理函数：比如sys\_sem\_open 中使用printf输出的信息不会到这个文件中，这时可以使用：./a.out | more来看。



**B**
*I*
[? Markdown 语法](#)

请输入想说的话

0 / 2000

发表评论

最新评论



Lighters\_c L10

楼上几位，TaskNode是在sleep\_on()里面被赋值的，sleep\_on()的原型是 void sleep\_on(struct task\_struct 星星p)，在sleep\_on()的函数体里面有一句 星p=current，current指向的就是当前任务(吐槽一句，实验楼的评论打星号好像有问题，就用汉字代替了)

2017-09-21 18:48:51

回复



zswlib L23

我也想知道，TaskNode成员什么时候设置的？？

2017-03-13 23:45:00

回复



codecat L42

学习！但是楼主的代码风格让人读起来比较痛苦，对着一句一句看完的，又自己写了一遍注释。

2017-01-30 21:40:20

回复



HUST\_AT L39

同学，请问“同时TaskNode成员指向的是等待消费这个信号的任务”，这个指针是在什么时候被赋值的。在sem\_open函数里面并没有看到设置TaskNode成员。但是到sem\_wait()和sem\_post()里面就直接阻塞或者唤醒TaskNode成员所指的进程了，那就是

说TaskNode成员肯定不是在这两个函数里面赋值的。但又没在sem\_open()里面看到对它的赋值，所以TaskNode成员到底什么时候指向了等待消费这个信号的任务。

2016-08-22 21:18:31

 回复



连接高校和企业



公司

- 关于我们
- 联系我们
- 加入我们

产品与服务

- 会员服务
- 蓝桥杯大赛
- 实战训练营
- 就业班
- 保入职

合作

- 1+X证书
- 高校实验教学
- 企业内训
- 合办学院
- 成为作者

学习路径

- Python学习路径
- Linux学习路径
- 大数据学习路径
- Java学习路径
- PHP学习路径
- 全部