WorstBrick L6 2017-02-25 20:51 Linux 计算机专业课 20 0

“操作系统原理与实践”实验报告

信号量的实现和应用

一.在内核中实现信号量

1.在POSIX标准中,sem_open()和sem_unlink()都是以字符串作为参数的,想要根据这个参数快速定位信号量在内核中的位置,自然就会想到用哈希表.我的想法是在OS启动后就初始化好一个全局的信号量的哈希表,表中的元素为struct sem_t *,初始化为NULL.哈希表的容量随便取了一个对本实验来说足够大素数.用最简单"线性探测法"来处理冲突,于是必须"懒惰删除"哈希表元素,详见注释

2.对于struct sem_t的内容,详见semaphore.h,我就说说挂起队列BlockQueue的实现:我没有选择指导书中方法———通过不同进程的内核栈上的局部变量形成的隐式链表,由sched.c中的sleep_on()和wake_up()来完成挂起和唤醒.我的方法比较笨,就是一个由链表实现的FIFO队列,写的很罗嗦,运行时开销也比较大———因为在一个信号量上阻塞或者唤醒进程会有很多对malloc()和free()的调用.

3.也因为2中的原因,我无法调用sleep_on()和wake_up()来改变当前进程current的状态(state).另一方面若直接对current用赋值语句,则要把struct task_struct的定义包进来,即还要include sched.h,太麻烦了,所以我在sched.c中写了个调用change_state()来专门做这件事.另外,为了操作方便,还在sem.c中写了一个系统调用void getsemval(struct sem_t * sem)来获取参数信号量当前的Value.

4.关于对error的处理,几乎是没有处理,反正指导书上也说这是一套缩水山寨版的POSIX信号量,就没有把那些过于庞杂的关于出错后擦屁股的东西加进来(而且我也不太懂),能把实验过了就行.

```
/*放在sched.c中*/
void change_state struct int
{
    tmp->state=new_state;
}
```

5.semaphore.h(内含本实验的山寨信号量所需的所有数据结构)

```
#ifndef SEMAPHORE_H_INCLUDED
#define SEMAPHORE_H_INCLUDED

/*Hash表的大小取一个质数*/
#define TABLE_SIZE 17

#ifndef NULL
#define NULL ((void *) 0)
#endif

/**
    链表的一个结点的定义如下: 一个域存放指向PCB的指针,
    一个域指向下一个链表结点
**/
typedef struct LNode * PtrToLNode;
struct LNode{
    struct task_struct * Process;
    PtrToLNode Next;
};

struct BQNode{
    PtrToLNode Head;
    PtrToLNode Rear;
};
typedef struct BQNode * BlockQueue;
/**
    (1)信号量的定义,包括名字(Name),当前的值(Value),阻塞在该信号量上的队列.
    对于这个队列BlockQueue,我打算做成一个FIFO链表:其中BQ是个指向结构体BQNode
    的指针,存放指向FIFO链表的头结点的指针Head,以及末尾节点的指针Rear.

    (2)因为用的是线性探测法处理哈希表的地址冲突,所以为sem_t分配了一个域Info,
    用来记录当前哈希表中该信号量的状态,当调用sem_unlink()删除信号量时进行"懒惰删除".
**/

typedef enum {Free,IsUsed,Deleted} Situation;
struct sem_t{
    Situation Info;
    char * Name;
    int Value;
    BlockQueue BQ;
};

/*模仿sched.h中的宏展开INIT_TASK,写一个用来初始化Hash表的宏*/

#define INIT_SEMHASH \
{NULL, NULL, NULL, NULL, NULL, NULL,\
NULL, NULL, NULL, NULL, NULL, NULL,\
NULL, NULL, NULL, NULL, NULL}

#endif
```

6.sem.c(内含本实验要实现的那几个山寨POSIX信号量的系统调用)

```
#include <semaphore.h>
#include <linux/kernel.h>
#include <asm/system.h>
#include <asm/segment.h>

extern void schedule void ;
extern void change_state struct int ;
extern struct task_struct * current;

/**
    因为是以信号量的名字来索引信号量,所以我打算用一个全局Hash表all_the_sem来存放信号量.
    Hash表中的元素为指向sem_t的指针,用一个宏INIT_SEMHASH来初始化
**/
struct sem_t * all_the_sem[TABLE_SIZE]=INIT_SEMHASH;


PtrToLNode NewLNode struct
{
    PtrToLNode newnode=(PtrToLNode) malloc (sizeof (struct LNode));
    newnode->Process=Process;
    newnode->Next=Next;

    return newnode;
}

/**
    以信号量的名字———字符串来索引信号量在Hash表中的位置
    用的是"移位法"———把字符串视为一个"32进制"的数,再将其转换成十进制,
    最后还要对Hash表的容量取模
**/
int SemHash const char int
{
    unsigned int H=0;
    char ch;
    int i=0;
/**
    一开始我直接用对Key解引用,即*Key来获得字符,结果发生Segment Fault,
    后来想起系统调用的那个实验中在内核中是不能直接取用户态空间中的数据的,
    必须通过asm/segment.h中的几个函数(包括get_fs_byte())来实现.
**/
    while ((ch=get_fs_byte(Key+i))!= '\0' )
    {
        H=(H<<5)+ch;
        i++;
    }

    return H%TableSize;
}

int FindSem const char int
{
    int position=SemHash(name,TABLE_SIZE);
    char ch;
    int i;

/**
    使用线性探测法来解决哈希地址冲突:
    (1)这里要说一下struct sem_t中的Info域.于使用"开放定址法"的哈希表都是用的"懒惰删除",
    即仅仅把sem->Info置为Free,而不是真的去删除它(这就是所谓的关闭信号量吗?).
    所以当有all_the_sem[position]->Info==Free,直接返回就行了
    (2)当sem->Info!=SameName时(即sem->Info==Free),说明有信号量正在占用哈希表中的这个位置,那么就用段功能类似于
    strcmp()的代码判断这个Free的信号量的Name与参数name是否相同,如果相同,将其Info设为SameName,即标这个
    信号量的Name是和参数name一样的(这样做的原因见sys_semopen()中的处理).如果不同,就看看其Info是否为Used,若是,说明有别的
    名字的信号量正在占着这个坑,就去找下一个位置呗;否则,即Info为Free,就选择该位置.
**/
    while (all_the_sem[position] && all_the_sem[position] ->Info!=SameName)
    {
        /**
            指导书上说使用string.h中的内嵌汇编处理字符串会破坏参数,为了保险起见就没去用,
            就自己实现了一段和strcmp()类似功能的代码:ch存放当前从地址(name+i)中取出的字符,
            和all_the_sem[position]->Name[i]对比.参数name和all_the_sem[position]->Name能够
            完全相同的唯一情况就是ch=='\0' && all_the_sem[position]->Name[i]=='\0'
        **/
        for (i=0;(ch=get_fs_byte(name+i))!= '\0' && all_the_sem[position] ->Name[i]!= '\0'
;i++)
        {
            if (all_the_sem[position] ->Name[i]!=ch)
                break ;
        }
        if (ch== '\0' && all_the_sem[position] ->Name[i]== '\0' )
        {
            all_the_sem[position] ->Info=SameName;
            break ;
        }
        else
        {
            if (all_the_sem[position] ->Info==Free)
                break ;
            else
                position=(position+ 1)%TableSize;
        }
    }

    return position;
}

/**
    创建信号量,或者打开一个已经存在的信号量的系统调用
**/
struct sem_t * sys_semopen const char int
{
    int position=FindSem(name,TABLE_SIZE);
    int Size=0;
    char ch;

    //Size中记录name中有几个字符(不包括结尾的'\0'),也是用get_fs_byte()
    while ((ch=get_fs_byte(name+Size))!= '\0' )
        Size++;

    //如果该信号量还不存在,就创建它
    if (all_the_sem[position]== NULL)
    {
        /**
            在这里我是选择把用户态中的字符串name中的内容通通复制到all_the_sem[position]->Name中,
            我也想过能不能直接令all_the_sem[position]->Name存放指向用户态字符串name的指针?感觉
            可行,不过就要对FindSem()进行改动了,以后再说吧.
        **/
        //先为struct sem_t分配所需的内存空间
        if ((all_the_sem[position]=( struct sem_t *) malloc (sizeof (struct sem_t )))== NULL)
            return NULL;
        all_the_sem[position] ->Name=NULL;
        all_the_sem[position] ->Info=Free;
        /**
            初始时,为挂起队列BQ分配一个无意义的头结点,令BQ->Head和BQ->Rear都指向它,
            为了方便FIFO链表的插入和删除.
        **/
        all_the_sem[position] ->BQ=(BlockQueue) malloc (sizeof (struct BQNode));
        all_the_sem[position] ->BQ->Head=NewLNode( NULL, NULL);
        all_the_sem[position] ->BQ->Rear=all_the_sem[position] ->BQ->Head;
    }
    /**
            Info==Free(即Info!=SameName)有两层含义:(1)这是一个新建的信号量,还没有Name.
            (2)它是一个曾经被关闭(懒惰删除)过的信号量,但它的Name和当前sys_semopen()的参数name不同.
            无论怎样,都要把它的名字变成参数name,只不过(2)中的情况还要把原来的Name释放掉
    **/
    if (all_the_sem[position] ->Info!=SameName)
    {
        if (all_the_sem[position] ->Name)
            free (all_the_sem[position] ->Name);
        all_the_sem[position] ->Name=( char *) malloc ((Size+ 1)* sizeof ( char ));
        for (;( int )Size>= 0;Size -- )
        {
            ch=get_fs_byte(name+Size);
            all_the_sem[position] ->Name[Size]=ch;
        }
    }
    /**
        如果前面的两个if条件都不满足,说明Info==SameName,这说明all_the_sem[position]->Name与
        参数name相等,直接设置好Info和Value就可以了
    **/
    all_the_sem[position] ->Info=IsUsed;
    all_the_sem[position] ->Value=value;

    return all_the_sem[position];
}
/**
    P原子操作的实现
**/
int sys_semwait struct sem_t
{
/**
    sem_wait()何时会出错?我觉得是参数sem不存在的时候,
    即参数sem==NULL时,
**/

    if (sem== NULL)
        return -1;

    cli();

    (sem ->Value) -- ;

    if (sem ->Value< 0)
    {
    /**
        首先,把当前进程current加入挂起队列.这是个FIFO的链表,在尾部(Rear)插入,
        在头部(Head)删除.
        然后调用change_state()改变当前进程的状态.
        最后调用schedule()切出去.
    **/
        sem->BQ->Rear ->Next=NewLNode(current, NULL);
        sem->BQ->Rear=sem ->BQ->Rear ->Next;
        change_state(sem ->BQ->Rear ->Process, 2);
        schedule();
    }

    sti();
    return 0;
}

int sys_sempost struct sem_t
{
    if (sem== NULL)
        return -1;
    cli();

    PtrToLNode tmp;

    (sem ->Value)++;
    /**
        这里我直接通过判断挂起队列是否为空来决定是否要唤醒进程,而不是
        if (sem->Value<=0)
    **/
    if ((tmp=sem ->BQ->Head ->Next)!= NULL)
    {
        change_state(tmp ->Process, 0);
    /**
        这里有个特殊情况要处理,就是挂起队列中只有一个进程的时候,当删除这个进程
        时,要把挂起队列恢复成初始化时的样子———重点是把sem->BQ->Rear重新指回
        那个无意义的头结点.我一开始没有进行这个处理,导致再往队列中添加进程时出错
        (因为是在尾部Rear添加新进程的)
    **/
        if ((sem ->BQ->Head ->Next=tmp ->Next)== NULL)
            sem->BQ->Rear=sem ->BQ->Head;
        free (tmp);
    }

    sti();
    return 0;
}

/**
    并没有研读过《UNIX环境高级编程》和POSIX标准,暂时无法理解
    为什么sem_unlink()要以信号量的名字,即字符串作参数.直接用struct sem_t * sem
    作参数不是更快更直接吗?
**/
int sys_semunlink const char
{
    int position=FindSem(name,TABLE_SIZE);
    PtrToLNode tmp,next;

    if (all_the_sem[position]== NULL)
        return -1;

/**
    对all_the_sem[position]进行懒惰删除,仅仅把Info置为Free,
    但也要把阻塞在该信号量上的进程都唤醒
**/
    all_the_sem[position] ->Info=Free;
    for (tmp=all_the_sem[position] ->BQ->Head ->Next;tmp;tmp=next)
    {
        next=tmp ->Next;
        change_state(tmp ->Process, 0);
        free (tmp);
    }
    all_the_sem[position] ->BQ->Head ->Process= NULL;
    all_the_sem[position] ->BQ->Head ->Next= NULL;
    all_the_sem[position] ->BQ->Rear=all_the_sem[position] ->BQ->Head;

    return 0;
}

/**
    为了操作方便,还写了一个系统调用getsemval()来获取参数信号量当前的Value
**/
int sys_getsemval struct sem_t
{
    return sem ->Value;
}
```

二.pc.c的实现

0.本人觉得pc.c的实现是本实验中最难的部分,整整花了我3天时间(整个实验我做了两个周末一共4天),究其原因,就是我在进程同步这一块掌握不牢,以及几个操作文件的POSIX接口用的太烂——— **说白了还是自己水平太垃圾** .奉劝想入坑的同学先把这两块知识学扎实了再做实验.

1.pc.c的难点在于怎么在文件这个共享缓冲区中实现实现produce和consume这两个动作.以下是我的尝试:

> 看到"共享",我一开始就想当然的只用一个文件描述符来打开这个文件,即consumer和producer都用同一个文件描述符来处理缓冲区,结果输出的消费序列是乱的(比如011234465这种).
> 于是拍脑门一想,"嗨呀既然使用同一个文件指针,那consumer和producer各自的文件偏移量肯定是一样的嘛,所以才会出错"于是就想通过信号量FreeBuffer(标示空闲的缓冲区的量)+ProductQuant(标示缓冲区中的产品数量)+lseek()来计算出consumer获得产品的正确位置,结果自然还是错的:由于OS对进程难以预测的调度,当有多个consumer时,是很有可能在ProductQuant上阻塞了多个进程的,即ProductQuant可能变成负数.于是,就算ProductQuant恢复回了正值,其值也无法正确反应当前缓冲区中产品的数量,它仅仅反映了执行sem_wait()而不被挂起的进程的数量而已.FreeBuffer也是同理.
> 最后就百度了一下,"多个进程共享一个文件"的正确方法,才想到让consumer和producer各自支配一个指向同一个文件的文件描述符,才实验成功.具体的做法见pc.c

2.因为产品的序列较大(>=500),所以决定把进程ID+产品号输出到一个文件Output中,而不是直接输出到Bochs的屏幕上.

3.同样的,没有对error进行处理.

4.我的这个pc.c有时会发生死锁,不知是程序逻辑的问题还是说这本来就是普遍现象?但无论死锁与否,都可以保证输出结果的正确,我也懒得去找原因了.

5.代码如下

```
#define __LIBRARY__
#include <unistd.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define CONSUMER_QUANT3
#define BUFFER_QUANT10
#define PRODUCT_RANGE501
#define CONSUME_RANGE500
_syscall2( struct sem_t *,sem_open, const char *,name, int ,value); /*sem_open的接口*/
_syscall1( int ,sem_wait, struct sem_t *,sem); /*sem_wait的接口*/
_syscall1( int ,sem_post, struct sem_t *,sem); /*sem_post的接口*/
_syscall1( int ,sem_unlink, const char *,name); /*sem_unlink的接口*/
_syscall1( int ,getsemval, struct sem_t *,sem); /*getsemval的接口*/


/**
    关于上面的那几个宏,从上往下分别表示:消费者的数量,缓冲区的数量,
    生产的产品的取值范围,消费的产品的取值范围
**/
void Producer int struct sem_t struct sem_t struct sem_t

{
    int Product= 0;

    while (Product<=PRODUCT_RANGE)
    {
        sem_wait(FreeBuffer);
        sem_wait(Mutex);
        /**
            为了保证Producer和Consumer可以在限定大小的缓冲区中活动,
            每当产品Product为BUFFER_QUANT的倍数时(说明Producer已经到了
            缓冲区的末尾了),就用lseek()把文件偏移量拨回到文件起始处
        **/
        if (Product%BUFFER_QUANT== 0)
            lseek(BufferI, 0,SEEK_SET);
        write(BufferI,( void *)(&Product), sizeof ( int ));
        Product++;
        sem_post(Mutex);
        sem_post(ProductQuant);
    }
}

void Consumer int int struct sem_t struct sem_t struct
 sem_t struct sem_t
{
    int Product,Length;
    char * Tmp=( char *) malloc ( 20 *sizeof ( char ));

    while ( 1 )
    {
        sem_wait(ProductQuant);
        /**
            把循环终止的判断放在这里是为了和
            结尾的while循环配套
        **/
        if (getsemval(ConsumeTimes)>CONSUME_RANGE)
            break ;
        sem_wait(Mutex);
        /**
            请联系Producer()中对文件偏移量的处理.对于Consumer,
            当读到文件尾时,就用lseek()把文件偏移量拨回到文件起始处.
        **/
        if (read(BufferO,( void *)(&Product), sizeof ( int ))== 0)
        {
            lseek(BufferO, 0,SEEK_SET);
            read(BufferO,( void *)(&Product), sizeof ( int ));
        }
        /**
            在缓冲区中Producer写入以及Consumer读入的都是int值,
            而在文件Output中打印出的消费记录自然是根据ASCII码
            显示,所以Consumer要往Output中写入char,这需要
            一个字符串Tmp做中转站.
            但这样做有一个问题:不知为何,把Tmp的内容用write()写入
            Output的话,字符串中的'\n'在Output中会变成奇怪的符号,
            用Vim打开会显示成'换行^@
        **/
        sprintf (Tmp, "%d: %d\n" ,getpid(),Product);
        Length= strlen (Tmp);
        write(Output,( void *)Tmp,(Length+ 1)* sizeof ( char ));
        sem_post(ConsumeTimes);
        sem_post(Mutex);
        sem_post(FreeBuffer);
    }

    /**
        关于这个while循环,是用来处理ConsumeTimes已经超过
        CONSUME_RANGE了还有Consumer阻塞在ProductQuant
        中出不来的情况,不过感觉好像没什么作用,该死锁时还是会死锁
    **/
    while (getsemval(ProductQuant)< 0)
        sem_post(ProductQuant);
}
int main void
{
    struct sem_t * Mutex;
    struct sem_t * FreeBuffer;
    struct sem_t * ProductQuant;
    /**
        信号量ConsumeTimes不是用来互斥,把它视为一个记录当前总的消费
        次数的全局变量就行了,用它来通知消费者是否要退出循环,于是对
        ConsumeTimes也不会有sem_wait()操作
    **/
    struct sem_t * ConsumeTimes;
    pid_t CurrentID;
    int i,BufferI,BufferO,Output;

    if ((Mutex=sem_open( "Mutex" , 1))== NULL)
        exit ( 0);
    if ((FreeBuffer=sem_open( "FreeBuffer" ,BUFFER_QUANT))== NULL)
        exit ( 0);
    if ((ProductQuant=sem_open( "ProductQuant" , 0))== NULL)
        exit ( 0);
    if ((ConsumeTimes=sem_open( "ConsumeTimes" , 0))== NULL)
        exit ( 0);

    /*建立缓冲区和输出的终端*/
    BufferI=open( "/usr/root/buffer" ,O_WRONLY|O_TRUNC|O_CREAT);
    BufferO=open( "/usr/root/buffer" ,O_RDONLY|O_TRUNC);
    Output=open( "/usr/root/output.log" ,O_RDWR|O_TRUNC|O_CREAT);

    /**
        fork()出CONSUMER_QUANT个子进程作为消费者,
        父进程作为生产者.
    **/
    for (i= 0;i<CONSUMER_QUANT;i++)
    {
        if ((CurrentID=fork())== 0)
            break ;
    }

    if (CurrentID== 0)
        Consumer(BufferO,Output,Mutex,FreeBuffer,ProductQuant,ConsumeTimes);
    else
    {
        Producer(BufferI,Mutex,FreeBuffer,ProductQuant);
        /**
            用一个循环来让父进程回收子进程
        **/
        while (wait( NULL)!= - 1)
            continue ;
        sem_unlink( "Mutex" );
        sem_unlink( "FreeBuffer" );
        sem_unlink( "ProductQuant" );
        sem_unlink( "ConsumeTimes" );
        close(BufferI);
        close(BufferO);
        close(Output);
    }

    return 0;
}
```

三.实验报告

1.执行效果变化很大,打印出的消费序列完全是乱的.因为没有信号量对临界区进行保护,加上OS对进程调度的不确定性,导致生产者和消费者们的动作都无法做到互斥.也就是说,在临界区中完成一个动作之前,它们都有可能被另一个进程抢占,于是,在缓冲区中的取出的数据自然都是错的.比如,消费者C准备准备在缓冲区的起始处取出产品0时,就被生产者P抢占,而P已经完成了一轮的生产,也处于缓冲区的起始处,它往这里写入一个11.如果有轮到C执行,那么C取出的产品就是11,而不是0.

2.不可行.因为这会导致死锁的发生.假如Producer调用P(Mutex),成功进入临界区,但被阻塞在Empty上,此时,它需要一个Consumer执行V(Empty)将其唤醒.然而,此时临界区中已经存在了一个Producer了,如果Consumer调用P(Mutex),就会阻塞在Mutex上,即其无法进入临界区.结果就是,两者都拥有对方需要的资源,但都被阻塞,无法释放自己拥有的资源,形成死锁.

0