

哈尔滨工业大学计算机科学与技术学院
2016年秋季学期《操作系统》

Lab4:信号量的实现与应用

姓名	学号	联系方式
匡盟盟	1143220116	kuangmeng@msn.com
樊晨霄	15S008199	18513534698

目 录

一、实验目的	1
二、实验内容	1
实验基本内容	1
用信号量解决生产者—消费者问题	1
实现信号量	1
生产者消费者问题的模型	2
三、实验过程	2
四、回答问题	5
实验心得	6
附录	6
A. sem.c	6
B. 用于Ubuntu中运行的pc.c	8
C. 用于linux-0.11中运行的pc.c	10

一、实验目的

- 加深对进程同步与互斥概念的认识；
- 掌握信号量的使用，并应用它解决生产者—消费者问题；
- 掌握信号量的实现原理。

二、实验内容

实验基本内容

1. 在Ubuntu下编写程序，用信号量解决生产者—消费者问题；
2. 在0.11中实现信号量，用生产者—消费者程序检验之。

用信号量解决生产者—消费者问题

在Ubuntu上编写应用程序“pc.c”，解决经典的生产者—消费者问题，完成下面的功能：

1. 建立一个生产者进程，N个消费者进程（ $N>1$ ）；
2. 用文件建立一个共享缓冲区；
3. 生产者进程依次向缓冲区写入整数0,1,2,...,M, $M\geq 500$ ；
4. 消费者进程从缓冲区读数，每次读一个，并将读出的数字从缓冲区删除，然后将本进程ID和数字输出到标准输出；
5. 缓冲区同时最多只能保存10个数。

一种可能的输出效果是：

```
10: 0
10: 1
10: 2
10: 3
10: 4
11: 5
11: 6
12: 7
10: 8
12: 9
12: 10
12: 11
12: 12
.....
11: 498
11: 499
```

其中ID的顺序会有较大变化，但冒号后的数字一定是从0开始递增加一的。

pc.c中将会用到sem_open()、sem_close()、sem_wait()和sem_post()等信号量相关的系统调用，请查阅相关文档。

实现信号量

Linux在0.11版还没有实现信号量，Linus把这件富有挑战的工作留给了你。如果能实现一套山寨版的完全符合POSIX规范的信号量，无疑是很有成就感的。但时间暂时不允许我们这么做，所以先弄一套缩水版的类POSIX信号量，它的函数原型和标准并不完全相同，而且只包含如下系统调用：

```
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);
sem_t是信号量类型，根据实现的需要自定义。
```

`sem_open()`的功能是创建一个信号量，或打开一个已经存在的信号量。

- `name`是信号量的名字。不同的进程可以通过提供同样的`name`而共享同一个信号量。如果该信号量不存在，就创建新的名为`name`的信号量；如果存在，就打开已经存在的名为`name`的信号量。
- `value`是信号量的初值，仅当新建信号量时，此参数才有效，其余情况下它被忽略。
- 当成功时，返回值是该信号量的唯一标识（比如，在内核的地址、ID等），由另两个系统调用使用。如失败，返回值是`NULL`。

`sem_wait()`就是信号量的P原子操作。如果继续运行的条件不满足，则令调用进程等待在信号量`sem`上。返回0表示成功，返回-1表示失败。

`sem_post()`就是信号量的V原子操作。如果有等待`sem`的进程，它会唤醒其中的一个。返回0表示成功，返回-1表示失败。

`sem_unlink()`的功能是删除名为`name`的信号量。返回0表示成功，返回-1表示失败。

在`kernel`目录下新建“`sem.c`”文件实现如上功能。然后将`pc.c`从Ubuntu移植到0.11下，测试自己实现的信号量。

生产者消费者问题的模型

```

Producer(){
    生产一个产品item;
    P(Empty); //空闲缓存资源
    P(Mutex); //互斥信号量
    将item放到空闲缓存中;
    V(Mutex);
    V(Full); //产品资源
}
Consumer(){
    P(Full);
    P(Mutex);
    从缓存区取出一个赋值给item;
    V(Mutex);
    V(Empty);
    消费产品item;
}

```

其中PV操作的含义如下：

执行P操作`P(S)`时信号量`S`的值减1，若结果不为负则`P(S)`执行完毕，否则执行P操作的进程暂停以等待释放。

执行V操作`V(S)`时，`S`的值加1，若结果不大于0则释放一个因执行`P(S)`而等待的进程。

生产者消费者问题中需要用到三个信号量，其中一个互斥信号量，两个同步信号量。在进行互斥操作的时候一定要注意，P、V操作是成对出现的，先进行P操作，进入临界区，访问临界资源，在进行V操作，出临界区。互斥信号量的初值一般是1。

进行同步操作的时候，要分析清进程间的制约关系。同一信号量的P、V操作也要成对出现，但是其P、V操作分别在不同的进程中。同步信号量的初值一般与资源的数目有关。

信号量值的含义：大于零时，表示可用的临界资源数目；等于零时，表示资源正好用完了；小于零时，表示系统中因请求该资源而被阻塞的进程数目。

三、实验过程

1. 编写`sem.c`文件，放入`linux-0.11/kernel/`目录下；
2. 修改`linux-0.11/kernel/system_call.s`文件，修改中断向量数：

```
nr_system_calls = 76
```

3. 在linux-0.11/include/unistd.h中加上系统调用，同时加上信号量结构体的定义：

```
/*新增的系统调用号*/
#define __NR_sem_open 72
#define __NR_sem_wait 73
#define __NR_sem_post 74
#define __NR_sem_unlink 75
struct queue{
    int front;
    int rear;
    struct task_struct *wait[Maxlength];
};
typedef struct queue queue;
struct sem_t{
    int value;
    int used;
    struct queue waitsem;
};
typedef struct sem_t sem_t;
```

4. 修改linux-0.11/include/linux/sys.h文件，加上：

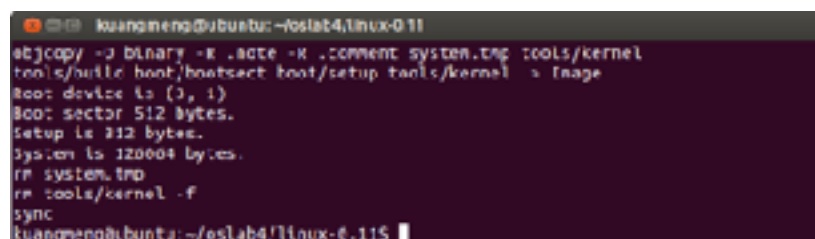
```
extern int sys_sem_open();
extern int sys_sem_wait();
extern int sys_sem_post();
extern int sys_sem_unlink();

, sys_sem_open, sys_sem_wait, sys_sem_post, sys_sem_unlink
```

5. 修改linux-0.11/kernel/Makefile，加上：

```
sem.o: sem.c ../include/errno.h ../include/signal.h
../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h
../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h
../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h
../include/asm/segment.h
```

6. 编译linux-0.11（进入linux-0.11目录下，直接“make all”即可）；



```
kuangmeng@ubuntu:~/oslab4/linux-0.11$ make all
objcopy -O binary -x .sect -K .comment system.tmp tools/kernel
tools/build boot/bootsect boot/setup tools/kernel > image
Boot device is (0, 1)
Boot sector 512 bytes.
Setup is 112 bytes.
System is 120004 bytes.
rm system.tmp
rm tools/kernel -f
sync
kuangmeng@ubuntu:~/oslab4/linux-0.11$
```

7. 编写pc.c文件，先放入oslab4/linux-0.11/目录下，使用指令“gcc -o pc pc.c -lpthread -Wall”进行编译并运行，命令及结果如下截图：

```

kuangmeng@ubuntu:~/oslab4
kuangmeng@ubuntu:~/oslab4$ gcc -D PC -D LPTHREAD_WAIT
pc.c: In function 'main':
pc.c:72:9: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
kuangmeng@ubuntu:~/oslab4$ ./pc
10554: 0
10554: 1
10554: 2
10554: 3
10554: 4
10554: 5
10554: 6
10554: 7
10552: 8
10552: 9
10553: 10
10553: 11
10553: 12
10553: 13
10553: 14
10553: 15
10554: 16

```

8. 再修改pc.c文件把系统调用的宏加上，再把sem_open的参数改一下，挂载linux-0.11之后，放入hdc/usr/root/目录下(同样需要修改hdc/usr/include/目录下的unistd.h文件，与之前的一致即可)，在bochs中编译运行，结果如图：

```

kuangmeng@ubuntu:~/oslab4
kuangmeng@ubuntu:~/oslab4$ cd oslab4
kuangmeng@ubuntu:~/oslab4$ sudo ./mount-hdc
[sudo] password for kuangmeng:
kuangmeng@ubuntu:~/oslab4$ ./run
mount hdc first
=====
Bochs x86 Emulator 2.3.7
build from cvs snapshot, on June 3, 2008
=====
000000000000i[ ] reading configuration from ./bochs/bochsrc.bxrc
000000000000i[ ] installing x module as the bochs GUI
000000000000i[ ] using log file ./bochsout.txt

```

在bochs中编译运行pc.c（由于bochs显示问题，我将输出打印到output.txt，同时使用vim打开截图分别如下）：

```

Bochs x86 emulator, http://bochs.sourceforge.net/
Bochs BIOS - build: 02-13-00
Revision: 1.194 $ $Date: 2007/12/23 19:46:27 $
Options: apmbios pcibios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)

Booting from floppy...

Loading system ...

Partition table ok.
37059/62000 free blocks
17518/20000 free inodes
3451 buffers = 3533824 bytes buffer space
Free mem: 12532912 bytes
ok.
(/usr/src/0118 gcc -o pc pc.c -Wall
pc.c: In function sem_open:
pc.c:8: warning: return of pointer from integer lacks a cast
(/usr/src/0118 ./pc > output.txt
(/usr/src/0118 ls
README      hello      linux-0.00  output.txt  pc.c
buffer.dat  hello.c    linux0.1gz  pc          sh0e
gcc11b140   hello.o    mtools.lnw  pc.c        sh0elace.tar.Z
(/usr/src/0118 vi output.txt
Ctrl + 3rd button enables mouse  R1: 0:0-1:0M  CPU: 30M

```



至此，该实验结束！

四、回答问题

1. 在pc.c中去掉所有与信号量有关的代码，再运行程序，执行效果有变化吗？为什么会这样？

答：有变化,输出没有按既定的顺序,甚至程序会崩溃。

原因：去掉了信号量有关的代码后，进程之间无法同步或者协作，这样就可能产生以下几种非正常情况：一种情况是缓冲区满了，生产者还在写入数据，这样会造覆盖掉没有被“消费”的一部分数据，以至于消费者“消费”的数据不是递增序列；另一种情况是缓冲区已经为空，消费者还尝试读取数据，读到的数据是已输出的数据（无效数据）；最后，由于多个进程对文件缓冲区同时访问，极易造成程序崩溃（出现fread()错误）。

2. 实验的设计者在第一次编写生产者——消费者程序的时候，是这么做的：

```

Producer(){
    P(Mutex); //互斥信号量
    生产一个产品item;
    P(Empty); //空闲缓存资源
    将item放到空闲缓存中;
    V(Full); //产品资源
    V(Mutex);
}
Consumer(){
    P(Mutex);
    P(Full);
    从缓存区取出一个赋值给item;
    V(Empty);
    消费产品item;
    V(Mutex);
}

```

这样可行吗？如果可行，那么它和标准解法在执行效果上会有什么不同？如果不可行，那么它有什么问题使它不可行？

答：这样做不可行。

原因：只有当缓冲区可写或者可读时，才能锁定该临界资源，否则容易出现缓冲区未锁定（mutex=1），consumer锁定该缓冲区，却发现empty=10，full=0，等待缓冲区有字符信号量，这样程序会产生饥饿并进入死锁状态；同理，当producer进入生产一个数据并锁定该缓冲区时，假设此时empty=0，mutex=0，P(Empty)操作之后，empty的值小于0，此时消费者进入等待信号量empty的等待队列上，可是由于mutex=0，此时并未解锁，两者都卡在等待mutex的状态。

实验心得

信号量的使用主要是用来保护共享资源，使得资源在一个时刻只有一个进程（线程）所拥有。信号量的值为正的时候，说明它空闲。所测试的线程可以锁定而使用它。若为0，说明它被占用，测试的线程要进入睡眠队列中，等待被唤醒。

通过这次实验，我们又更深层次的理解了老师课堂上讲的信号量的知识，也实践了没有信号量或者信号量实现不正确会是什么样子。体会到没有信号量就可能出现进程之间为争抢互斥资源而产生死锁的危害及解决办法。

附录

A. sem.c

```
#define __LIBRARY__
#include <unistd.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>
#include <asm/system.h>
#define SEM_COUNT 32
sem_t semaphores[SEM_COUNT];
/*队列相关操作，rear始终是下一个待写入的位置，front始终是队列第一个元素*/
void init_queue(sem_queue* q){
    q->front = q->rear = 0;
}
int is_empty(sem_queue* q){
    return q->front == q->rear?1:0;
}
/*留下标QUE_LEN-1不用，判断是否慢*/
int is_full(sem_queue* q){
    return (q->rear+1)%QUE_LEN == q->front?1:0;
}
/*获得队列头第一个任务*/
struct task_struct* get_task(sem_queue* q){
    if(is_empty(q)){
        printk("Queue is empty!\n");
        return NULL;
    }
    struct task_struct* tmp = q->wait_tasks[q->front];
    q->front = (q->front+1)%QUE_LEN;
    return tmp;
}
/*任务插入队列尾*/
int insert_task(struct task_struct* p, sem_queue* q){
    // printk("Insert %d", p->pid);
    if(is_full(q)){
        printk("Queue is full!\n");
        return -1;
    }
}
```



```

    }
    q->wait_tasks[q->rear] = p;
    q->rear = (q->rear+1)%QUE_LEN;
    return 1;
}
/*信号量是否已打开，是返回位置*/
int sem_location(const char* name){
    int i;

    for(i = 0; i < SEM_COUNT; i++){
        if(strcmp(name,semaphores[i].name) == 0 && semaphores[i].occupied == 1){
            return i;
        }
    }
    return -1;
}
/*打开信号量*/
sem_t* sys_sem_open(const char* name,unsigned int value){
    char tmp[16];
    char c;
    int i;
    for( i = 0; i<16; i++){
        c = get_fs_byte(name+i);
        tmp[i] = c;
        if(c =='\0') break;
    }
    if(c >= 16){
        printk("Semaphore name is too long!");
        return NULL;
    }
    if((i = sem_location(tmp)) != -1){
        return &semaphores[i];
    }
    for(i = 0; i< SEM_COUNT; i++){
        if(!semaphores[i].occupied){
            strcpy(semaphores[i].name,tmp);
            semaphores[i].occupied = 1;
            semaphores[i].value = value;
            init_queue(&(semaphores[i].wait_queue));
            // printk("%d %d %d
%s\n",semaphores[i].occupied,i,semaphores[i].value,semaphores[i].name);
            // printk("%p\n",&semaphores[i]);
            return &semaphores[i];
        }
    }
    printk("Numbers of semaphores are limited!\n");
    return NULL;
}
/*P原子操作*/
int sys_sem_wait(sem_t* sem){
    cli();
    sem->value--;
    if(sem->value < 0){
        /*参见sleep_on*/
        current->state = TASK_UNINTERRUPTIBLE;

```

```

        insert_task(current,&(sem->wait_queue));
        schedule();
    }
    sti();
    return 0;
}

```

/*V原子操作*/

```
int sys_sem_post(sem_t* sem){
```

```

    cli();
    struct task_struct *p;
    sem->value++;
    if(sem->value <= 0){
        p = get_task(&(sem->wait_queue));
        if(p != NULL){
            (*p).state = TASK_RUNNING;
        }
    }
    sti();
    return 0;
}

```

/*释放信号量*/

```

int sys_sem_unlink(const char *name){
    char tmp[16];
    char c;
    int i;
    for( i = 0; i<16; i++){
        c = get_fs_byte(name+i);
        tmp[i] = c;
        if(c == '\0') break;
    }
    if(c >= 16){
        printk("Semaphore name is too long!");
        return -1;
    }
    int ret = sem_location(tmp);
    if(ret != -1){
        semaphores[ret].value = 0;
        strcpy(semaphores[ret].name,"\0");
        semaphores[ret].occupied = 0;
        return 0;
    }
    return -1;
}

```

B. 用于Ubuntu中运行的pc.c

```

#define __LIBRARY__
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
int input_num(FILE* fp, int put_pos,int num){
    fseek( fp, put_pos*sizeof(int) , SEEK_SET );

```

```

    int run_code=fwrite( &num, 1, sizeof(num), fp);
    fflush(fp);
    return run_code;
}
int output_num(FILE* fp, int get_pos,int* num){
    fseek( fp, get_pos*sizeof(int) , SEEK_SET );
    return fread( num, sizeof(int),1, fp);
}
}

int main(){
    int const NUM=100;
    int const PRONUM=5;
    int const MAXSIZE=10;
    int i,j,k;
    int cost_num;
    int get_pos = 0;
    int put_pos = 0;
    sem_t *empty, *full, *mutex;
    FILE *fp = NULL;
    empty =(sem_t *)sem_open("empty",O_CREAT,0064,10);
    full  =(sem_t *)sem_open("full",O_CREAT,0064,0);
    mutex =(sem_t *)sem_open("mutex",O_CREAT,0064,1);
    fp=fopen("filebuffer.txt", "wb+");
    input_num(fp,10,get_pos);
    if( !fork() ){
        for( i = 0 ; i < NUM; i++){
            sem_wait(empty);
            sem_wait(mutex);
            input_num(fp,put_pos,i);
            put_pos = ( put_pos + 1)% MAXSIZE;
            sem_post(mutex);
            sem_post(full);
        }
        exit(0);
    }
    for( k = 0; k < PRONUM ; k++ ){
        if( !fork() ){
            for( j = 0; j < NUM/PRONUM; j++ ){
                sem_wait(full);
                sem_wait(mutex);
                fflush(stdout);
                output_num(fp,10,&get_pos);
                output_num(fp,get_pos,&cost_num);
                printf("%d:  %d\n",getpid(),cost_num);
                fflush(stdout);
                get_pos = (get_pos + 1) % MAXSIZE;
                input_num(fp,10,get_pos);
                sem_post(mutex);
                sem_post(empty);
            }
            exit(0);
        }
    }
    wait(NULL);
    sem_unlink("empty");
    sem_unlink("full");
}

```

```
    sem_unlink("mutex");
    fclose(fp);
    return 0;
}
```

C. 用于linux-0.11中运行的pc.c

```
#define __LIBRARY__
#include <unistd.h>

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
_syscall2(sem_t*,sem_open,const char *,name,unsigned int,value);
_syscall1(int,sem_wait,sem_t*,sem);
_syscall1(int,sem_post,sem_t*,sem);
_syscall1(int,sem_unlink,const char *,name);

#define NUMBER 520 /*打出数字总数*/
#define CHILD 5 /*消费者进程数*/
#define BUFSIZE 10 /*缓冲区大小*/
sem_t *empty, *full, *mutex;
int fno; /*文件描述符*/
int main(){
    int i,j,k;
    int data;
    pid_t p;
    int buf_out = 0; /*从缓冲区读取位置*/
    int buf_in = 0; /*写入缓冲区位置*/
    /*打开信号量*/
    if((mutex = sem_open("carmutex",1)) == SEM_FAILED){
        perror("sem_open() error!\n");
        return -1;
    }
    if((empty = sem_open("carempty",10)) == SEM_FAILED){
        perror("sem_open() error!\n");
        return -1;
    }
    if((full = sem_open("carfull",0)) == SEM_FAILED){
        perror("sem_open() error!\n");
        return -1;
    }
    fno = open("buffer.dat",O_CREAT|O_RDWR|O_TRUNC,0666);
    /* 将待读取位置存入buffer后,以便 子进程 之间通信 */
    lseek(fno,10*sizeof(int),SEEK_SET);
    write(fno,(char *)&buf_out,sizeof(int));
    /*生产者进程*/
    if((p=fork())==0){
        for( i = 0 ; i < NUMBER; i++){
            sem_wait(empty);
            sem_wait(mutex);
            /*写入一个字符*/
            lseek(fno, buf_in*sizeof(int), SEEK_SET);
            write(fno,(char *)&i,sizeof(int));
```

```

        buf_in = ( buf_in + 1)% BUFSIZE;
        sem_post(mutex);
        sem_post(full);
    }
    return 0;
}else if(p < 0){
    perror("Fail to fork!\n");
    return -1;
}
}
for( j = 0; j < CHILD ; j++ ){
    if((p=fork())==0){
        for( k = 0; k < NUMBER/CHILD; k++ ){
            sem_wait(full);
            sem_wait(mutex);
            /*获得读取位置*/
            lseek(fno,10*sizeof(int),SEEK_SET);
            read(fno,(char *)&buf_out,sizeof(int));
            /*读取数据*/
            lseek(fno,buf_out*sizeof(int),SEEK_SET);
            read(fno,(char *)&data,sizeof(int));
            /*写入读取位置*/
            buf_out = (buf_out + 1) % BUFSIZE;
            lseek(fno,10*sizeof(int),SEEK_SET);
            write(fno,(char *)&buf_out,sizeof(int));
            sem_post(mutex);
            sem_post(empty);
            /*消费资源*/
            printf("%d: %d\n",getpid(),data);
            fflush(stdout);
        }
        return 0;
    }else if(p<0){
        perror("Fail to fork!\n");
        return -1;
    }
}
}
wait(NULL);
/*释放信号量*/
sem_unlink("carfull");
sem_unlink("carempty");
sem_unlink("carmutex");
/*释放资源*/
close(fno);
return 0;
}

```