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](mailto:kuangmeng@msn.com) |
|  |  |  |

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

附录 6

1. sem.c 6
2. 用于Ubuntu中运行的pc.c 8
3. 用于linux-0.11中运行的pc.c 9

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>=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。

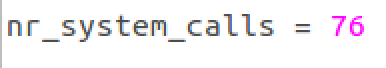
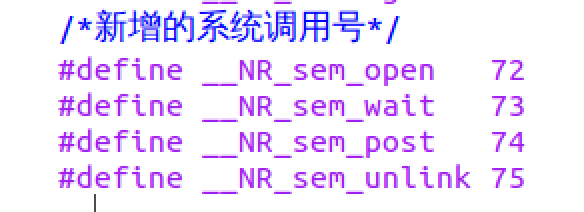
执行V操作V(S)时，S的值加1，若结果不大于0则释放一个因执行P(S)而等待的进程。

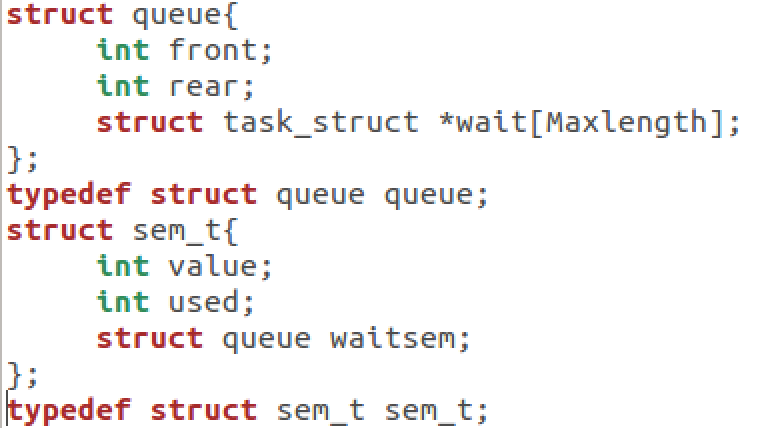
生产者消费者问题中需要用到三个信号量，其中一个互斥信号量，两个同步信号量。在进行互斥操作的时候一定要注意，P、V操作是成对出现的，先进行P操作，进入临界区，访问临界资源，在进行V操作，出临界区。互斥信号量的初值一般是1.

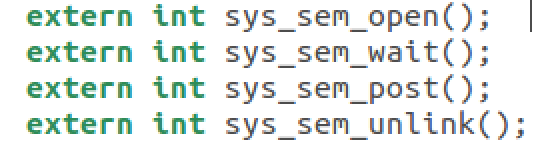
进行同步操作的时候，要分析清进程间的制约关系。同一信号量的P、V操作也要成对出现，但是其P、V操作分别在不同的进程中。同步信号量的初值一般与资源的数目有关。

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。

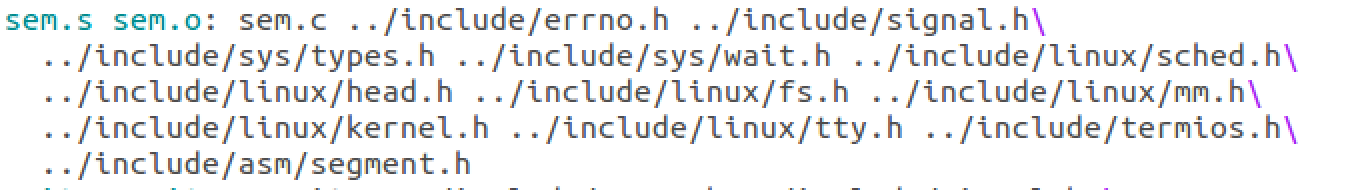
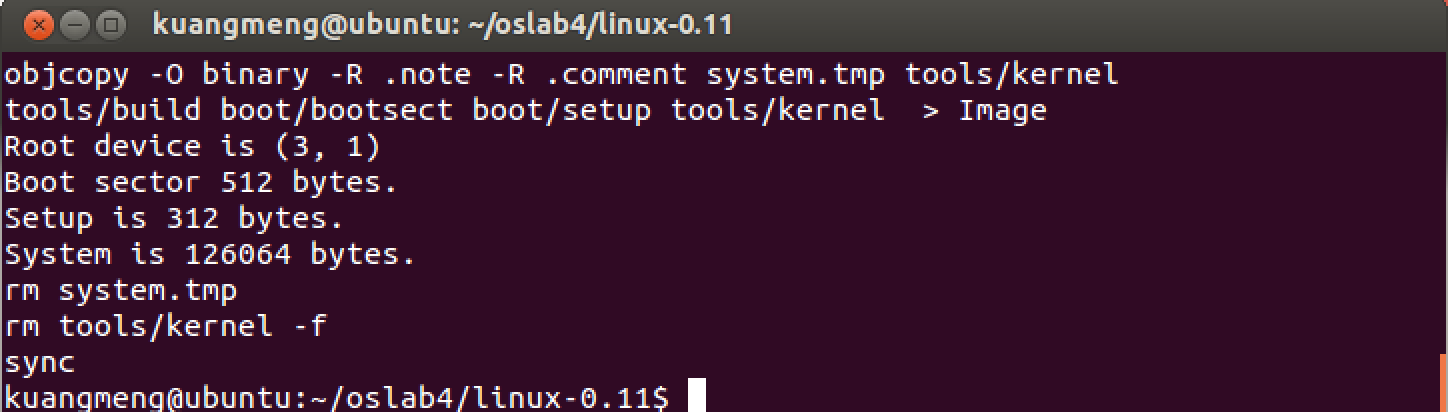
三、实验过程

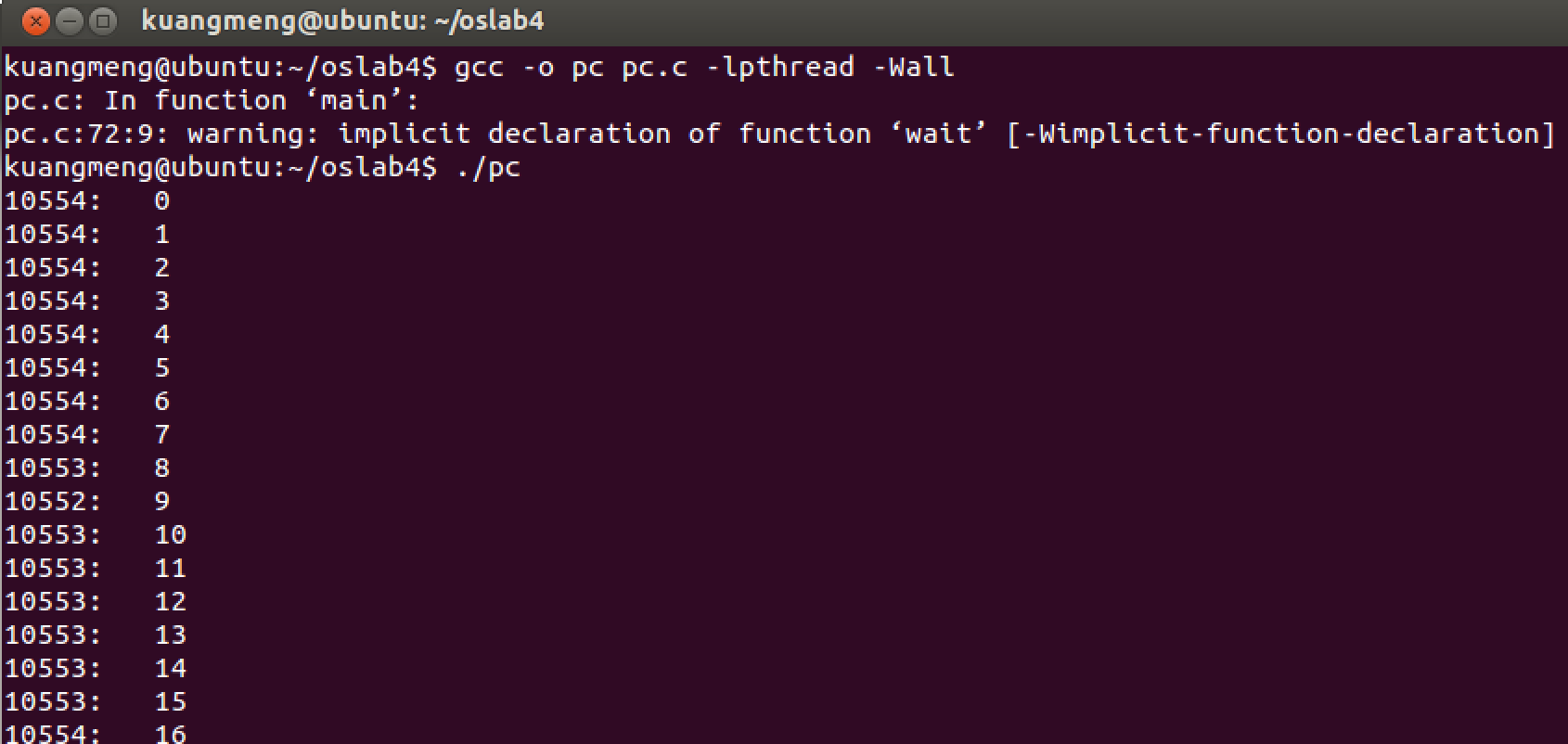
* 1. 编写sem.c文件，放入linux-0.11/kernel/目录下；
  2. 修改linux-0.11/kernel/system\_call.s文件，修改一个值：
  3. 在linux-0.11/include/unistd.h中加上系统调用，同时加上信号量结构体的定义：



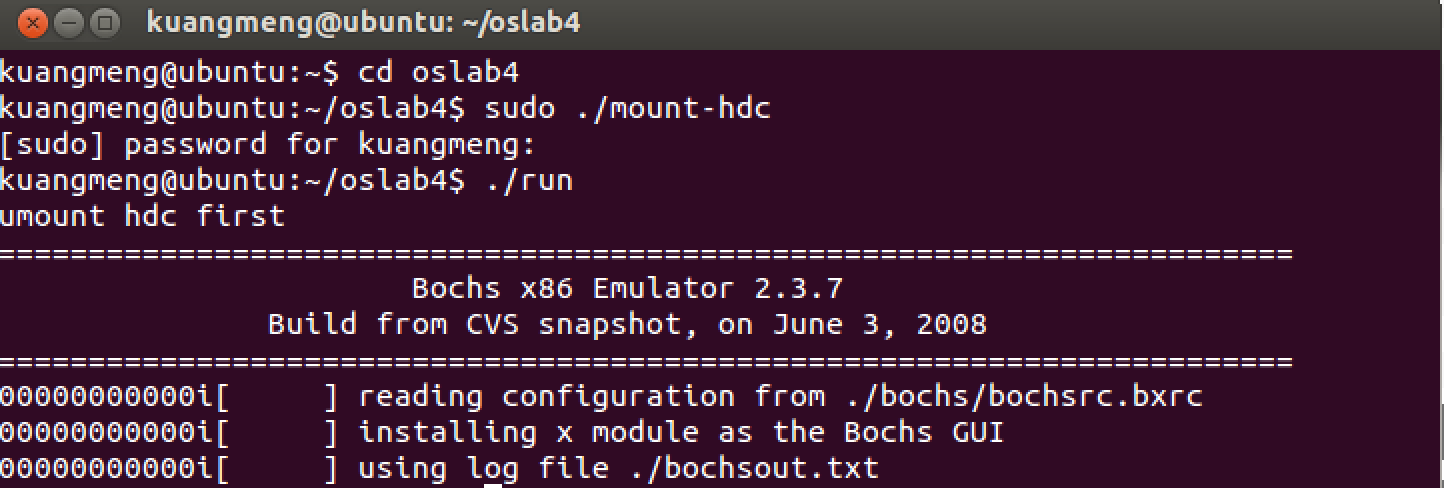
* 1. 修改linux-0.11/include/linux/sys.h文件，加上：



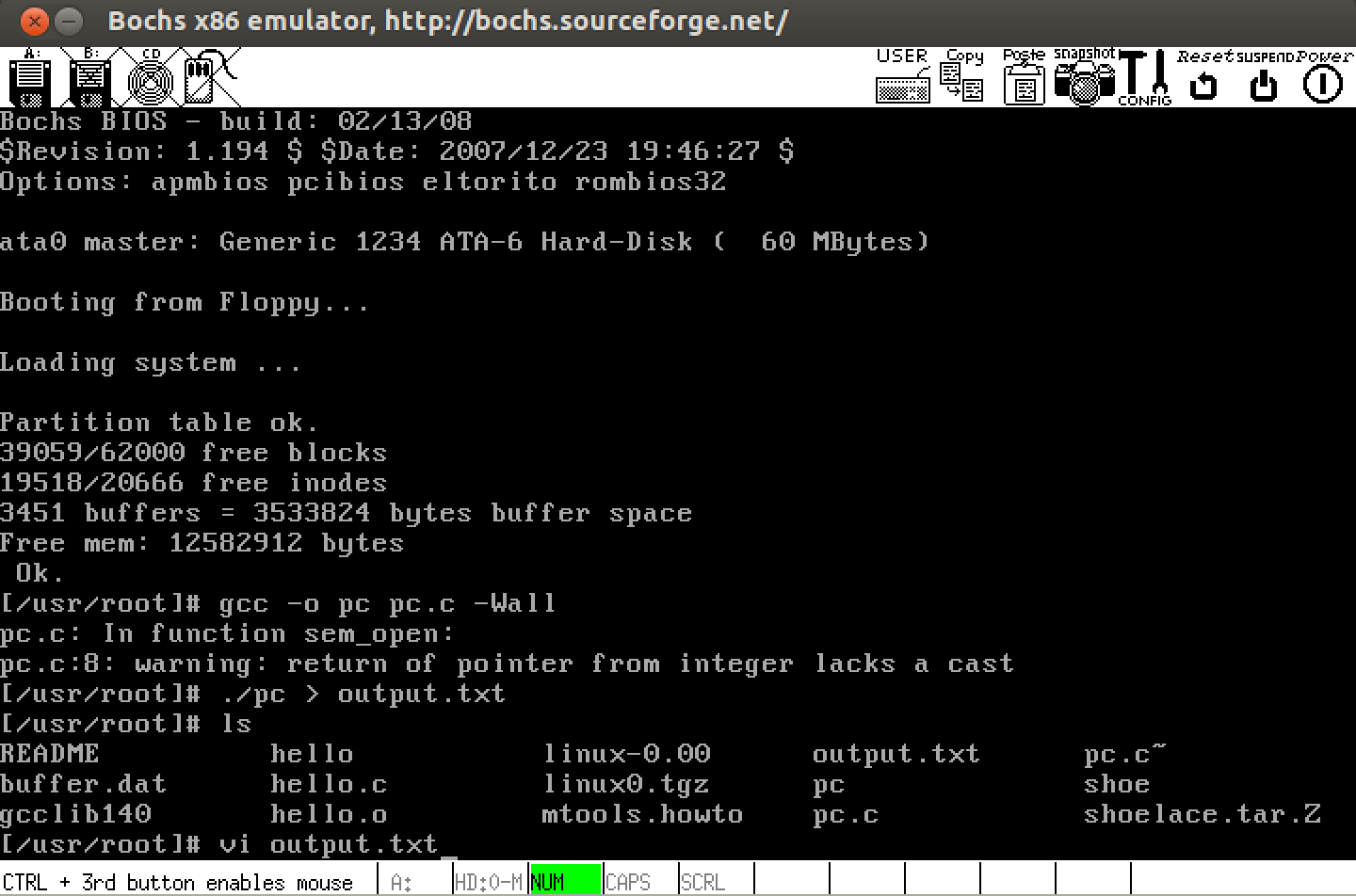
* 1. 修改linux-0.11/kernel/Makefile，加上：
  2. 编译linux-0.11（进入linux-0.11目录下，直接“make all”即可）；
  3. 编写pc.c文件，先放入oslab4/linux-0.11/目录下，使用指令“gcc -o pc pc.c -lpthread -Wall”进行编译并运行，命令及结果如下截图:



* 1. 再修改pc.c文件把系统调用的宏加上，再吧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：



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）：





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,输出没有顺序,甚至程序会崩溃。

原因：没有了信号量，进程之间无法同步或者协作，一种情况是缓冲区满了，生产者还在写入数据，会造覆盖掉部分数据。或者缓冲区为空，消费者尝试读取数据，读到的数据是已输出的数据。同时，由于多个进程对文件缓冲区同时访问，极容易造成程序奔溃。

1. 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，这样程序会进入死锁状态。

附录

1. 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("Semphore 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;

}

1. 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;

}

1. 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;

}