Linux 内核实验报告

实验题目：构造新内核同步机制实验

学号： 200900301236 （辅修号： ）
日期：2012.5.11 班级： 09 软 1 姓名：王添枝
Email：tzwang2012@163.com

实验目的：设计同步原语模拟内核信号量。

硬件环境：
软件环境：ubuntu 10.10
linux 内核:2.6.35.13
gcc:4.4.5

实验步骤：

一、实验设计

信号量负责对共享缓冲区的互斥，主要实现的方法有信号量的声明（sys_sema_open），在特定信号量的等待（sys_sema_wait），将特定信号量上的等待进程唤醒（sys_sema_signal），删除特定信号量（sys_sema_close）。

声明信号量 ：声明新的信号量结构，对本信号量等待队列初始化，本信号量信号灯数目初始化，然后将其放入信号量等待队列中。

在特定信号量的等待 ：如果信号灯数目大于 0，说明还有资源可以利用。否则，说明没有资源可用，**prepare_to_wait_exclusive(tmp->p,&wait,TASK_INTERRUPTIBLE)** 放入等待队列，此函数使用 signal 唤醒时只唤醒一个进程。调用 schedule（），重新调度其他进程 。Schedule（）方法后是 finish_wait() 方法，当进程被唤醒后，首先执行此方法。此方法将进程真正地从等待队列中弹出。

唤醒信号量上的等待进程 ：进程即将从缓冲区离开，将信号灯的数目+1。此时，如果等待队列中还有等待进程（判断条件是信号灯的数目<0），则从等待队列中弹出一个进程，使其状态变为 RUNNING，等待被调度。

删除特定信号量 ：给出信号量的编号，判断等待队列上是否还有等待的进程，如果有的话，将所有的进程均从队列中弹出。然后从信号量的队列中删除编号是输入数字的信号量结构。

二、数据结构

```
typedef struct __sema{
int key;//本信号量编号
int number;//本信号量信号灯个数
wait_queue_head_t *p; // 系统等待队列首指针
struct __sema *next;//所有信号量以链形式串在一起，本属性指向下一信号量结构
}sema;
```

三、自定义函数

1. **sema * check_sema_key(int key,sema **prev)**

作用：检查以 key 值为编号的信号量是否存在于当前的信号量链中。

2.**asmlinkage int sys_sema_open(int key,int number)**

作用：如果 key=0 声明一个新的信号量（链表的插入），如果 key!=0,返回 key 值。

3.**asmlinkage int sys_sema_wait(int key)**

作用： 如果信号灯数目大于 0，说明还有资源可以利用，将信号灯数目减 1。否则，说明没有资源可用， prepare_to_wait_exclusive(tmp->p,&wait,TASK_INTERRUPTIBLE)放入等待队列，调用 schedule（），重新调度其他进程。Schedule（）方法后是 finish_wait() 方法，当进程被唤醒后，首先执行此方法。此方法将进程真正地从等待队列中弹出。

4. **asmlinkage int sys_sema_signal(int key)**

作用：进程即将从缓冲区离开，将信号灯的数目+1。此时，如果等待队列中还有等待进程（判断条件是信号灯的数目<0），则从等待队列中弹出一个进程，使其状态变为 RUNNING，等待被调度。

5.**int wake_up_all_ones(int key)**

作用：唤醒 key 编号信号量上所有的等待进程。

三、调试记录

1. open 0 1

其中的 0 表示创建一个新的，1 表示初始信号量为 1

2. wait 2

其中 2 为 open 所创建的标识符，由于信号量为 1，大于 0，因此不会等待

3. wait 2

由于信号量已经变为 0，因此会在等待

4. 另启一个命令行，执行 wait 2

同样的在等待

5. 再启一个命令行，执行 signal 2

此时将会唤醒上面两个在等待中的一个，再 signal 2 将会唤醒另一个进程。

6. 执行 partB,将会启动 5 个进程，交替的读取中断次数。

注意：本程序中存在一个漏洞，在运行完后，如果都 close，若再 open 则进程会被杀死。如果 close 一部分，没有全都 close，则再 open 不会被杀死。

四、结论分析与体会

由于上次编译内核编译了很久，此次精简了内核，精简完后不能链接无线，也没有声音，但是编译速度快了很多。

此次实验遇到的主要问题是不知道 singal 唤醒时如何让它只唤醒一个进程。还有一个没解决的就是上面提到的都 close 后再 open 进程会被杀死。

通过此次实验，对内核同步机制有了一定的了解。

程序完整源代码：

1. 在 ipc/shm.c 中添加的代码

```
typedef struct  __sema{
   int key;//本信号量编号
   int number;//本信号量信号灯个数
   wait_queue_head_t *p; // 系统等待队列首指针
   struct __sema *next;//所有信号量以链形式串在一起，本属性指
向下一信号量结构
}sema;

sema * sema_head = NULL;
sema * sema_end = NULL;
/*
* 检查以key值为编号的信号量是否存在于当前的信号链中
*/
sema * check_sema_key(int key,sema **prev){
   sema *tmp = sema_head;
   *prev = NULL;
   while(tmp){
      if(tmp->key==key)
         return tmp;
      *prev = tmp;
      tmp = tmp->next;
   }
   return NULL;
}
/**
*如果key=0声明一个新的信号量，如果key!=0,返回key值
*/
asmlinkage int sys_sema_open(int key,int number){
   sema *prev;
   sema *new;
   if(key){
      if(!check_sema_key(key,&prev))
         return -1;
      else
         return key;
   }
   else
   {
      printk("sys_sema_open create a new sema\n");
      new =(sema *)kmalloc(sizeof(sema),GFP_KERNEL);
      new->p=(wait_queue_head_t
*)kmalloc(sizeof(wait_queue_head_t),GFP_KERNEL);
      new->next=NULL;
      init_waitqueue_head(new->p);
```

```
        //new->p.task_list.next = &new->p.task_list;
        //new->p.task_list.prev = &new->p.task_list;
        new->number=number;
        printk("sys_sema_open center\n");
        if(!sema_head){
           printk("sys_sema_open first start\n");
           new->key=2;//从2开始按偶数递增事件号
           sema_head = sema_end = new;
           printk("sys_sema_open create first\n");
           return new->key;
        }
        else
        {//事件队列不为空，按偶数递增一个事件号
           printk("sys_sema_open not first start\n");
           new->key = sema_end->key +2;
           sema_end->next = new;
           sema_end = new;
           printk("sys_sema_open create not first\n");

        }
        printk("sys_sema_open success\n");
        return new->key;
    }
}

asmlinkage int sys_sema_wait(int key)
{
  sema *tmp;
  sema *prev = NULL;
  //取出指定事件的等待队列头指针
   tmp = check_sema_key(key,&prev);
  if(tmp!=NULL&&tmp->number<=0){
    tmp->number--;
    printk("sys_sema_wait\n");
     DEFINE_WAIT(wait); //初始化等待队列入口
       //使调用进程进入阻塞状态
                              prepare_to_wait_exclusive(tmp-
>p,&wait,TASK_INTERRUPTIBLE);
     schedule(); //引发系统重新调度
     finish_wait(tmp->p,&wait); //等待进程被唤醒，结束阻塞状
态
     return key;
  }
   else if(tmp!=NULL){
```

```
      printk("sys_sema_wait greater than 0\n");
      tmp->number--;
      return key;
   }
  return -1;
}

//唤醒在指定事件上等待的进程的系统调用：
asmlinkage int sys_sema_signal(int key)
{
   sema *tmp = NULL;
   sema *prev = NULL;

  //取出指定事件的等待队列头指针
   tmp = check_sema_key(key,&prev);
   if(tmp==NULL)
      return 0;
   tmp->number++;
   if(tmp->number<=0){
      wake_up(tmp->p);   //唤醒它
      printk("sys_sema_signal wake_up\n");
   }
   printk("sys_sema_signal over\n");
     return 1;
}

/*
*  唤醒key编号信号量上所有的等待进程。
*/
int wake_up_all_ones(int key){
   sema *tmp = NULL;
   sema *prev = NULL;
   tmp = check_sema_key(key,&prev);
   if(tmp==NULL)
      return 0;
   if(tmp->number<0){
      wake_up_all(tmp->p);//wake_up_all(wait_queue_head_t
*queue);        wake_up_interruptible_all(wait_queue_head_t
*queue); 这种 wake_up 唤醒所有的进程, 不管它们是否进行独占等待
(可中断的类型仍然跳过在做不可中断等待的进程)
      printk("wake_up_all_ones\n");
   }
   return 1;
}
```

```
asmlinkage int sys_sema_close(int key)
{
  sema *prev=NULL;
  sema *releaseItem;

  if((releaseItem = check_sema_key(key,&prev)) != NULL){
   //找到指定事件
    if( releaseItem == sema_end) //在队尾
      sema_end = prev;
    else if(releaseItem == sema_head) //在队首
      sema_head = sema_head->next;
    else  //在队中
      prev->next = releaseItem->next;
   int jieshou = wake_up_all_ones(key);//如果阻塞则唤醒它
   if(jieshou == 0)
      printk("no such sempahore\n");
   if(releaseItem){
      printk("sys_sema_close\n");
       kfree(releaseItem); //释放事件节点
       return 1;
   }
  }
  return 0;
}
```

2.open.c

```
#include <stdio.h>
#include <stdlib.h>
#include
"/lib/modules/2.6.35.13/build/arch/x86/include/asm/unistd_3
2.h"

int sema_open(int key,int number){
  return syscall(__NR_sema_open,key,number);
}

int main(int argc, char ** argv)
{
  int i;
  if(argc != 3)
  {
     printf("Usage: open 2|4|6|2i semaNum \n");
    return -1;
  }
  i = sema_open(atoi(argv[1]),atoi(argv[2]));
```

```
  if(i)
          printf("Registed event %d .\n",i);
  else
    printf("Registed event fail !\n");
  return 0 ;
}
```

3.wait.c

```
#include <stdio.h>
#include <stdlib.h>
#include
"/lib/modules/2.6.35.13/build/arch/x86/include/asm/unistd_3
2.h"

int sema_wait(int num){
  return syscall(__NR_sema_wait,num);
}

int main(int argc, char ** argv)
{
  int i;
  if(argc != 2)
  {
     printf("Usage: wait 2 | 4 | 6 | 2i ...\n");
    return -1;
  }
  printf("Event %d will sleeping !\n",atoi(argv[1]));
  i = sema_wait(atoi(argv[1]));
  printf("Event %d wakeup !\n",i);
  return 0 ;

}
```

4.signal.c

```
#include <stdio.h>
#include <stdlib.h>
#include
"/lib/modules/2.6.35.13/build/arch/x86/include/asm/unistd_3
2.h"

int sema_signal(int num){
  return syscall(__NR_sema_signal,num);
}

int main(int argc, char ** argv)
{
```

```
  int i;
  if(argc != 2)
  {
     printf("Usage: signal 2 | 4 | 6 | 2i ...\n");
    return -1;
  }

  i = sema_signal(atoi(argv[1]));
  if(i)
    printf("Wakeup event %d .\n",atoi(argv[1]));
  else
    printf("No event %d !\n",atoi(argv[1]));
  return 0 ;
}
```

5.close.c

```
 #include <stdio.h>
#include <stdlib.h>
#include
"/lib/modules/2.6.35.13/build/arch/x86/include/asm/unistd_3
2.h"
int sema_close(int flag){
  return syscall(__NR_sema_close,flag);
}
int main(int argc, char ** argv)
{
  int i;
  if(argc != 2)
    return -1;
  i = sema_close(atoi(argv[1]));
  if(i==1)
   printf("Unregisted event %d\n",atoi(argv[1]));
  else
   printf("Unregisted error.");
  return 0 ;
}
```

6.partB.c

```
#include <stdio.h>
#include <stdlib.h>
#include
"/lib/modules/2.6.35.13/build/arch/x86/include/asm/unistd_3
2.h"

int sema_open(int key,int number){
  return syscall(__NR_sema_open,key,number);
```

```
}
int sema_close(int flag){
  return syscall(__NR_sema_close,flag);
}
int sema_signal(int num){
  return syscall(__NR_sema_signal,num);
}
int sema_wait(int num){
  return syscall(__NR_sema_wait,num);
}
long interrupt_num(){
  return syscall(__NR_mysyscall1);
}
int main(int argc, char ** argv)
{
  int i;
  int pid[5];
  i=sema_open(0,1);
  printf("register %d\n",i);
  int m;
  for(m=0;m<5;m++){
   if((pid[m]=fork())==0){//子进程
      while(1){
         printf("process %d is waiting \n",getpid());
         int j= sema_wait(i);
         //printf("wait over %d\n",j);
         sleep(2);
         long intr = interrupt_num();
         printf("process      %d      interrupt      time:
%ld\n",getpid(),intr);
         int k= sema_signal(i);
         printf("signal %d\n",i);
      }
      //return 0;
   }else
      continue;
  }
 //int m= sema_close(i);
// printf("wait over %d\n",m);
  return 0 ;

}
```

参考材料

linux 操作系统内核实习
linux 内核设计与实现