OS lab4实验报告

姓名	学号	邮箱	院系
张洋彬	191220169	1016466918@qq.com	计算机科学与技术系

一、实验目的

- 实现格式化输入函数
- 完成4个子例程：`syscallSemInit`、`syscallSemWait`、`syscallSemPost`和`syscallSemDestroy`
- 解决进程同步问题

二、实验预备知识

2.1信号量

`P()`（Prolaag，荷兰语尝试减少）

- sem减1
- 如sem<0，进入等待，否则继续

`V()`（Verhoog，荷兰语增加）

- sem加1
- 如sem<=0，唤醒一个等待进程

信号量的实现（伪代码）：

```
class Semaphore {
  int sem;
  WaitQueue q;
}

Semaphore::P(){
  sem--;
  if(sem < 0){
    Add this thread t to q;
    block(t)
  }
}

Semaphore::V(){
  sem++;
  if(sem <= 0){
    Remove a thread t from q;
    wakeup(t);
  }
}
```

2.2进程的阻塞和取出阻塞

这样将current线程加到信号量i的阻塞列表可以通过以下代码实现

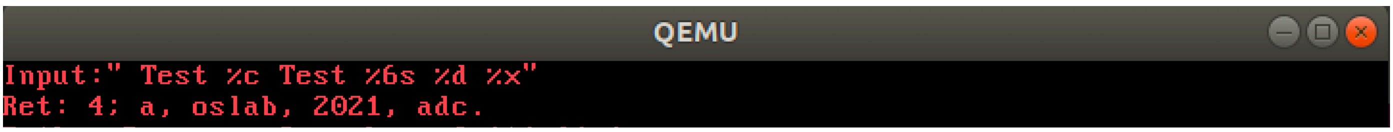
```
  pcb[current].blocked.next = sem[i].pcb.next;
  pcb[current].blocked.prev = &(sem[i].pcb);
  sem[i].pcb.next = &(pcb[current].blocked);
  (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
```

以下代码可以从信号量i上阻塞的进程列表取出一个进程：

```
  pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
        (uint32_t)&(((ProcessTable*)0)->blocked));
  sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
  (sem[i].pcb.prev)->next = &(sem[i].pcb);
```

三、实验结果

输入`Test a Test oslab 2021 0xadc`，结果如下图所示：



完成四个信号量函数后，结果如下：

```
Input:" Test %c Test %6s %d %x"
Ret: 4; a, oslab, 2021, 12.
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
```

可见，同时只允许两个“进程”进入缓冲区

哲学家思考问题：

生产者消费者问题如下：

读者写者问题

```
reader_writer
Writer 0: write
Reader 0: read, total 1 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
Writer 1: write
Writer 2: write
Writer 0: write
Reader 0: read, total 1 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
Reader 0: read, total 2 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
Writer 1: write
Writer 2: write
Writer 0: write
Reader 0: read, total 1 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
```

四、实验过程

4.1 实现 syscallReadStdIn 和 keyboardHandle

`keyboardHandle`要做的事情就两件：

1、将读取到的`keyCode`放入到`keyBuffer`中

2、唤醒阻塞在`dev[STD_IN]`上的一个进程

代码如下：

```
void keyboardHandle(struct StackFrame *sf) {
  ProcessTable *pt = NULL;
  uint32_t keyCode = getKeyCode();
  if (keyCode == 0) // illegal keyCode
    return;
  //putChar(getChar(keyCode));
  keyBuffer[bufferTail] = keyCode;
  bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;

  if (dev[STD_IN].value < 0) { // with process blocked
      唤醒阻塞在dev[STD_IN]上的一个进程
  }

```

```
  return;
}
```

`syscallReadStdIn`要做的事情也就两件：

1、如果`dev[STD_IN].value == 0`，将当前进程阻塞在`dev[STD_IN]`上

2、进程被唤醒，读`keyBuffer`中的所有数据

值得注意的就是最多只能有一个进程被阻塞在`dev[STD_IN]`上，多个进程想读，那么后来的进程会返回`-1`，其他情况`scanf`的返回值应该是实际读取的字节数

代码如下：

```
void syscallReadStdIn(struct StackFrame *sf) {
  if(dev[STD_IN].value<0){
    pcb[current].regs.eax=-1;
    return;
  }
  else if(dev[STD_IN].value==0){
    dev[STD_IN].value--;
    阻塞当前进程
    asm volatile("int $0x20");
    读keyBuffer中的所有数据
    pcb[current].regs.eax=keybuffer中读取的字节数;
    return;
  }
}

```

4.2 实现信号量

4.2.1 实现 sem_init

sem_init系统调用用于初始化信号量，其中参数`value`用于指定信号量的初始值，初始化成功则返回`0`，指针`sem`指向初始化成功的信号量，否则返回-1

伪代码逻辑如下：

```
for(遍历sem数组):
  如果找到未使用的信号量:
    修改信号量为使用状态
    修改信号量的value为传入的参数 (edx)
    初始化next和prev
    return 未使用的信号量的下标;
  else :
    return -1;
```

4.2.2 实现 sem_post

sem_post系统调用对应信号量的`V`操作，其使得`sem`指向的信号量的`value`增一，若`value`取值不大于`0`，则释放一个阻塞在该信号量上进程（即将该进程设置为就绪态），若操作成功则返回`0`，否则返回`-1`

伪代码如下：

```
void syscallSemPost(struct StackFrame *sf) {
  int i = (int)sf->edx;
  if (i < 0 || i >= MAX_SEM_NUM) {//下标超限
    pcb[current].regs.eax = -1;
    return;
  }
  if(sem[i].state==1){
    sem[i].value++;
    if(sem[i].value<=0){
      释放一个阻塞的进程
      更改阻塞进程的状态为STATE_RUNNABLE
    }
    pcb[current].regs.eax=0;//成功
    return;
  }
  else{
    pcb[current].regs.eax = -1;
    return;
  }
}
```

4.2.3 实现 sem_wait

sem_wait系统调用对应信号量的`P`操作，其使得`sem`指向的信号量的`value`减一，若`value`取值小于`0`，则阻塞自身，否则进程继续执行，若操作成功则返回`0`，否则返回`-1`

```
void syscallSemWait(struct StackFrame *sf) {
  int i=sf->edx;
  if (i < 0 || i >= MAX_SEM_NUM) {
    pcb[current].regs.eax = -1;
    return;
```

```
  }
  if(sem[i].state==1){
    sem[i].value--;
    if(sem[i].value<0){
      将自身阻塞;
      pcb[current].state=STATE_BLOCKED;
      pcb[current].sleepTime=-1;
      asm volatile("int $0x20");//陷入时钟中断
    }
    pcb[current].regs.eax=0;
    return;
  }
  else{
    pcb[current].regs.eax=-1;
  }
  return;
}
```

4.2.4 实现 sem_destroy

`sem_destroy`系统调用用于销毁`sem`指向的信号量，销毁成功则返回`0`，否则返回`-1`，若尚有进程阻塞在该信号量上，可带来未知错误，若有进程堵塞也返回-1。

```
void syscallSemDestroy(struct StackFrame *sf) {
  int i = (int)sf->edx;
  if (i < 0 || i >= MAX_SEM_NUM) {
    pcb[current].regs.eax = -1;
    return;
  }
  if(sem[i].state==1){
    if(sem[i].value>=0){
      sem[i].state=0;
      pcb[current].regs.eax=0;
      return;
    }
    else{//有进程堵塞
      pcb[current].regs.eax=-1;
      return;
    }
  }
  else{
    pcb[current].regs.eax=-1;
    return;
  }
}
```

4.3 解决进程同步问题

4.3.1 哲学家用餐问题

思路如下：

```c
#define N 5                    // 哲学家个数
semaphore fork[5];             // 信号量初值为1
void philosopher(int i){       // 哲学家编号：0-4
  while(TRUE){
    think();   // 哲学家在思考
    sleep(128);
    if(i%2==0){
      P(fork[i]);
      sleep(128);// 去拿左边的叉子
      P(fork[(i+1)%N]);
      sleep(128);// 去拿右边的叉子
    } else {
      P(fork[(i+1)%N]);        // 去拿右边的叉子
      sleep(128);
      P(fork[i]);              // 去拿左边的叉子
      sleep(128);
    }
    eat();
    sleep(128);// 吃面条
    V(fork[i]);
    sleep(128);// 放下左边的叉子
    V(fork[(i+1)%N]);
    sleep(128);// 放下右边的叉子
  }
}
```

没有死锁，可以实现多人同时就餐，任意P、V及思考、就餐动作之间添加sleep(128)。

循环创建五个子进程，子进程陷入while（1）循环里，不会创建多余的进程。

代码如下：

```c
int uEntry(void) {

  for(int i=0;i<N;i++){
    sem_init(&forks[i],1);
  }

  for(int i=0;i<N;i++){
    int ret=fork();
    if(ret==0){
      while(1){
        philosopher(i);
```

```
        }
        break;
      }

    }
    while(1);
    for(int i=0;i<N;i++){
      sem_destroy(&forks[i]);
    }
    return 0;
```

4.3.2 生产者与消费者问题

假设缓冲区大小为3，代码参考实验教程，创建4个子进程运行producer，父进程运行consumer

```
for(int i=0;i<4;i++){
    if(fork()==0){
      producer(i);
    }
}
consumer();
```

4.3.3 读者写者问题

大致过程如伪代码，创建三个读进程，三个子进程，为了实现进程间通信（针对 Rcount 变量），还使用了系统调用 write() 和 read() 。

五、感想与心得

本次实验难度适中，实验介绍里给了很多本实验会涉及的代码,完成下来觉得对于信号量的理解更深了，本次实验也用到了双向链表，之前有点没看懂，（有点蠢了）但问了同学之后，也搞明白了进程的阻塞和取出阻塞，关于信号量这方面已经大概懂了，只剩最后一个文件实验了，冲冲冲！