

# OS lab4实验报告

姓名	学号	邮箱	院系
张洋彬	191220169	<a href="mailto:1016466918@qq.com">1016466918@qq.com</a>	计算机科学与技术系

## OS lab4实验报告

- 一、实验目的
- 二、实验预备知识
  - 2.1信号量
  - 2.2进程的阻塞和取出阻塞
- 三、实验结果
- 四、实验过程
  - 4.1 实现 syscallReadStdIn 和 keyboardHandle
  - 4.2 实现信号量
    - 4.2.1 实现 `sem_init`
    - 4.2.2 实现 `sem_post`
    - 4.2.3 实现 `sem_wait`
    - 4.2.4 实现 `sem_destroy`
  - 4.3 解决进程同步问题
    - 4.3.1 哲学家用餐问题
    - 4.3.2 生产者与消费者问题
    - 4.3.3 读者写者问题
- 五、感想与心得

## 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）：

```
1 class Semaphore {
2     int sem;
3     WaitQueue q;
4 }
5
6 Semaphore::P(){
7     sem--;
8     if(sem < 0){
9         Add this thread t to q;
10        block(t)
11    }
12 }
13
14 Semaphore::V(){
15     sem++;
16     if(sem <= 0){
17         Remove a thread t from q;
18         wakeup(t);
19     }
20 }
```

## 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

```
1 pcb[current].blocked.next = sem[i].pcb.next;
2 pcb[current].blocked.prev = &(sem[i].pcb);
3 sem[i].pcb.next = &(pcb[current].blocked);
4 (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：

```
1 pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
2     (uint32_t)&((ProcessTable*)0)->blocked));
3 sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
4 (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：

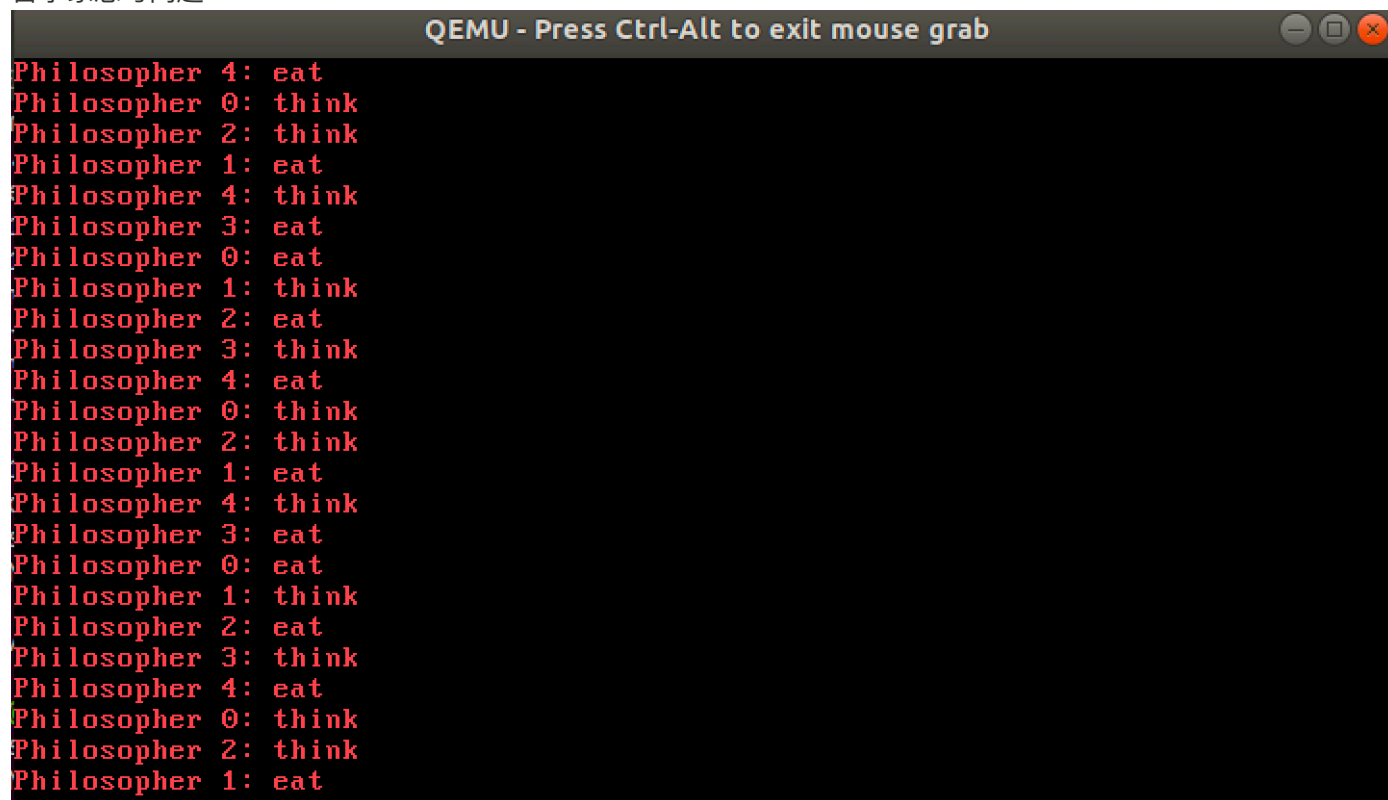


完成四个信号量函数后，结果如下：

```
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

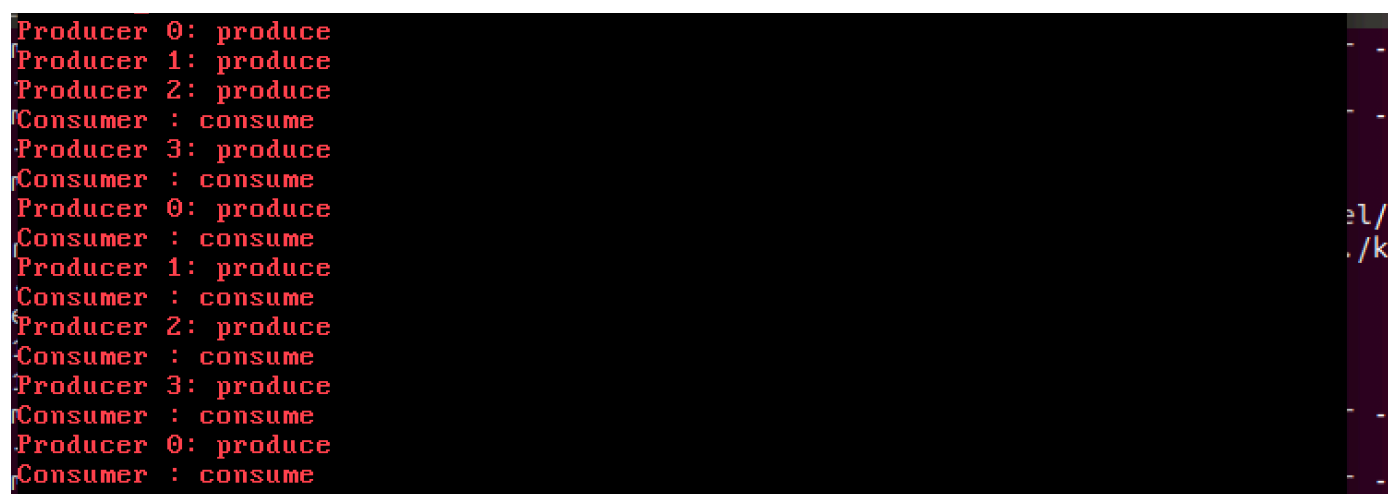
哲学家思考问题：



The screenshot shows a terminal window titled "QEMU - Press Ctrl-Alt to exit mouse grab". The terminal displays a sequence of actions for five philosophers (0 to 4). Each philosopher alternates between "eat" and "think" states. The output is as follows:

```
Philosopher 4: eat
Philosopher 0: think
Philosopher 2: think
Philosopher 1: eat
Philosopher 4: think
Philosopher 3: eat
Philosopher 0: eat
Philosopher 1: think
Philosopher 2: eat
Philosopher 3: think
Philosopher 4: eat
Philosopher 0: think
Philosopher 2: think
Philosopher 1: eat
Philosopher 4: think
Philosopher 3: eat
Philosopher 0: eat
Philosopher 1: think
Philosopher 2: eat
Philosopher 3: think
Philosopher 4: eat
Philosopher 0: think
Philosopher 2: think
Philosopher 1: eat
```

生产者消费者问题如下：



The screenshot shows a terminal window displaying a sequence of "produce" and "consume" actions for four producers (0 to 3) and four consumers. The output is as follows:

```
Producer 0: produce
Producer 1: produce
Producer 2: produce
Consumer : consume
Producer 3: produce
Consumer : consume
Producer 0: produce
Consumer : consume
Producer 1: produce
Consumer : consume
Producer 2: produce
Consumer : consume
Producer 3: produce
Consumer : consume
Producer 0: produce
Consumer : consume
```

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：

```
1 void keyboardHandle(struct StackFrame *sf) {
2     ProcessTable *pt = NULL;
3     uint32_t keyCode = getKeyCode();
4     if (keyCode == 0) // illegal keyCode
5         return;
6     //putChar(getChar(keyCode));
7     keyBuffer[bufferTail] = keyCode;
8     bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;
9
10    if (dev[STD_IN].value < 0) { // with process blocked
11        唤醒阻塞在dev[STD_IN]上的一个进程
12    }
13 }
```

```
14     return;
15 }
```

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：

```
1 void syscallReadStdIn(struct StackFrame *sf) {
2     if(dev[STD_IN].value<0){
3         pcb[current].regs.eax=-1;
4         return;
5     }
6     else if(dev[STD_IN].value==0){
7         dev[STD_IN].value--;
8         阻塞当前进程
9         asm volatile("int $0x20");
10        读keyBuffer中的所有数据
11        pcb[current].regs.eax=keybuffer中读取的字节数;
12        return;
13    }
14 }
15
```

## 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：

```

1  for(遍历sem数组):
2      如果找到未使用的信号量:
3          修改信号量为使用状态
4          修改信号量的value为传入的参数 (edx)
5          初始化next和prev
6          return 未使用的信号量的下标;
7  else :
8      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：

```

1  void syscallSemPost(struct StackFrame *sf) {
2      int i = (int)sf->edx;
3      if (i < 0 || i >= MAX_SEM_NUM) { //下标超限
4          pcb[current].regs.eax = -1;
5          return;
6      }
7      if(sem[i].state==1){
8          sem[i].value++;
9          if(sem[i].value<=0){
10             释放一个阻塞的进程
11             更改阻塞进程的状态为STATE_RUNNABLE
12         }
13         pcb[current].regs.eax=0; //成功
14         return;
15     }
16     else{
17         pcb[current].regs.eax = -1;
18         return;
19     }
20 }

```

## 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

```

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

```

```

6     }
7     if(sem[i].state==1){
8         sem[i].value--;
9         if(sem[i].value<0){
10            将自身阻塞;
11            pcb[current].state=STATE_BLOCKED;
12            pcb[current].sleepTime=-1;
13            asm volatile("int $0x20");//陷入时钟中断
14        }
15        pcb[current].regs.eax=0;
16        return;
17    }
18    else{
19        pcb[current].regs.eax=-1;
20    }
21    return;
22 }

```

#### 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。

```

1 void syscallSemDestroy(struct StackFrame *sf) {
2     int i = (int)sf->edx;
3     if (i < 0 || i >= MAX_SEM_NUM) {
4         pcb[current].regs.eax = -1;
5         return;
6     }
7     if(sem[i].state==1){
8         if(sem[i].value>=0){
9             sem[i].state=0;
10            pcb[current].regs.eax=0;
11            return;
12        }
13        else{//有进程堵塞
14            pcb[current].regs.eax=-1;
15            return;
16        }
17    }
18    else{
19        pcb[current].regs.eax=-1;
20        return;
21    }
22 }

```



## 4.3 解决进程同步问题

### 4.3.1 哲学家用餐问题

思路如下：

```
1  #define N 5                // 哲学家个数
2  semaphore fork[5];         // 信号量初值为1
3  void philosopher(int i){    // 哲学家编号: 0-4
4      while(TRUE){
5          think(); // 哲学家在思考
6          sleep(128);
7          if(i%2==0){
8              P(fork[i]);
9              sleep(128); // 去拿左边的叉子
10             P(fork[(i+1)%N]);
11             sleep(128); // 去拿右边的叉子
12         } else {
13             P(fork[(i+1)%N]); // 去拿右边的叉子
14             sleep(128);
15             P(fork[i]); // 去拿左边的叉子
16             sleep(128);
17         }
18         eat();
19         sleep(128); // 吃面条
20         V(fork[i]);
21         sleep(128); // 放下左边的叉子
22         V(fork[(i+1)%N]);
23         sleep(128); // 放下右边的叉子
24     }
25 }
```

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：

```
1  int uEntry(void) {
2
3      for(int i=0;i<N;i++){
4          sem_init(&forks[i],1);
5      }
6
7      for(int i=0;i<N;i++){
8          int ret=fork();
9          if(ret==0){
10             while(1){
11                 philosopher(i);
```

```
12     }
13     break;
14 }
15
16 }
17 while(1);
18 for(int i=0;i<N;i++){
19     sem_destroy(&forks[i]);
20 }
21 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

```
1  for(int i=0;i<4;i++){
2      if(fork()==0){
3          producer(i);
4      }
5  }
6  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！