

# 2022spring 《操作系统》lab4实验报告

姓名：武雅琛

学号：201220102

exercise: 4 task:3 😊

## 一、PART1: exercise

### exercise1

- 请回答一下，什么情况下会出现死锁。
  - 如果五个哲学家同时拿起了左手边的叉子，他们就都会进入P操作等待右手边的叉子。但是由于所有哲学家都进入了等待状态，所以没有一个哲学家可以通过V操作释放一把叉子，导致任何人都吃不到通心粉，无限等待右手边的叉子。
  - 也就是五个进程都执行到了  $P((fork[i]+1)\%N)$ ; 这行语句等待V操作。

### exercise2

- 说一下该方案有什么不足？（答出一点即可）

```
1  #define N 5                // 哲学家个数
2  semaphore fork[5];         // 信号量初值为1
3  semaphore mutex;           // 互斥信号量，初值1
4  void philosopher(int i){   // 哲学家编号：0-4
5  while(TRUE){
6      think();               // 哲学家在思考
7      P(mutex);              // 进入临界区
8      P(fork[i]);             // 去拿左边的叉子
9      P(fork[(i+1)%N]);       // 去拿右边的叉子
10     eat();                  // 吃面条
11     V(fork[i]);             // 放下左边的叉子
12     V(fork[(i+1)%N]);       // 放下右边的叉子
13     V(mutex);              // 退出临界区
14 }
15 }
```

- 互斥变量的P操作应该在其他信号量之后，否则可能在后续的P操作中占用互斥锁，陷入无限等待。

### exercise3

- 正确且高效的解法有很多，请你利用信号量PV操作设计一种正确且相对高效（比方案2高效）的哲学家吃饭算法。（其实网上一堆答案，主要是让大家多看看不同的实现。）
  - 思路一：如果只有少于等于四名哲学家取用叉子吃面条，那么很容易推断无论如何都不会发生死锁，所以可以增加一个信号量count来记录正在取用叉子和进食的哲学家的人数。

```
1  #define N 5                // 哲学家个数
2  semaphore fork[5];         // 信号量初值为1
3  semaphore count;           // 可以进食的哲学家数量，初始值为4
```

```

4 void philosopher(int i){ // 哲学家编号: 0-4
5 while(TRUE){
6     think(); // 哲学家在思考
7     P(count); // 根据进餐人数判断当前是否可以进餐
8     P(fork[i]); // 去拿左边的叉子
9     P(fork[(i+1)%N]); // 去拿右边的叉子
10    eat(); // 吃面条
11    V(fork[i]); // 放下左边的叉子
12    V(fork[(i+1)%N]); // 放下右边的叉子
13    P(count); // 进餐结束后, 释放一个可进餐资源
14 }
15 }

```

- **思路二**: 发生死锁是因为哲学家在同时取用左手边的叉子的时候没有意识到冲突, 所以我们可以让**冲突更早的出现**用以判断。

具体的实现方法是对编号后的哲学家为**奇数的拿起右手边的叉子**, **偶数的哲学家拿起左手边的叉子**, 这样也可以规避五个哲学家同时拿起一侧的叉子。

```

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     if (i % 2) // 如果编号为偶数
7         P(fork[(i+1)%N]); // 去拿右边的叉子
8         P(fork[i]); // 去拿左边的叉子
9     else // 如果编号为奇数
10        P(fork[i]); // 去拿左边的叉子
11        P(fork[(i+1)%N]); // 去拿右边的叉子
12    eat(); // 吃面条
13    V(fork[i]); // 放下左边的叉子
14    V(fork[(i+1)%N]); // 放下右边的叉子
15    P(count); // 进餐结束后, 释放一个可进餐资源
16 }
17 }

```

## exercise4

- **为什么要用两个信号量呢? emptyBuffers和fullBuffer分别有什么直观含义?**
  - emptyBuffers初始化为n, 指的是Buffer中可以**被生产者写入**的空间。
  - fullBuffers初始化为0, 指的是Buffer中已经被生产者写入, 可以**被消费者消费**的空间。
  - 因为在信号量与PV操作中, 只有当**信号量小于等于零**的时候执行P操作**进程被挂起等待资源释放**。
  - 所以当实现两个进程同步的时候需要两个信号量来代表上述的**两类资源**。
  - 如果只使用一个信号量, 比如说fullBuffers, 那么**生产者无法意识到Buffer空间耗尽挂起等待**, 而是继续生产数据, 这和生产者-消费者问题中对生产者和消费者的行为和Buffer的行为规定是矛盾的。反之只使用emptyBuffers同理。
  - 并且如果只使用一个信号量, 那么在操作**同一个信号量**的时候必然要加上**互斥锁**防止对信号量发生冲突 (count)。比如采用了fullBuffer, 如果消费者使得fullBuffer为0, 等待资源释放, 但是没有释放互斥锁, 那么生产者和消费者都会陷入无尽的等待当中。

## 三、PART2:task

## task1

- 完成格式化输入

完成syscallReadStdIn函数

☒ `dev[**STD_IN**].value==0`，设备忙碌，进程阻塞

将进程挂载到读设备的阻塞列表上：

```
1 dev[STD_IN].value = dev[STD_IN].value - 1;
2     pcb[current].blocked.next = dev[STD_IN].pcb.next;
3     pcb[current].blocked.prev = &(dev[STD_IN].pcb);
4     dev[STD_IN].pcb.next = &(pcb[current].blocked);
5     (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
```

触发时钟中断处理进程调度：

```
1 pcb[current].state = STATE_BLOCKED;
2     pcb[current].sleepTime = 2147483647;
3     asm volatile("int $0x20");
```

进程被唤醒后读取字符缓存中的内容并传入用户内存空间，这里要注意获取用户数据段的段选择子来保证访问到正确的用户内存地址

```
1 int sel = sf->ds;
2     char *str = (char *)sf->edx;
3     asm volatile("movw %0, %%es"::"m"(sel));
4     int i = 0;
5     //int len = (bufferTail - bufferHead + MAX_KEYBUFFER_SIZE)
6     % MAX_KEYBUFFER_SIZE;
7     while(bufferHead != bufferTail && i < sf->ebx)
8     {
9         char character = keyBuffer[bufferHead];
10        putString("obatin character:");
11        putchar( keyBuffer[bufferHead]);
12        putchar('\n');
13        asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"
14        (str + i));
15        ++i;
16        bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
17    }
18    //enableInterrupt();
```

返回读取的字符数目：

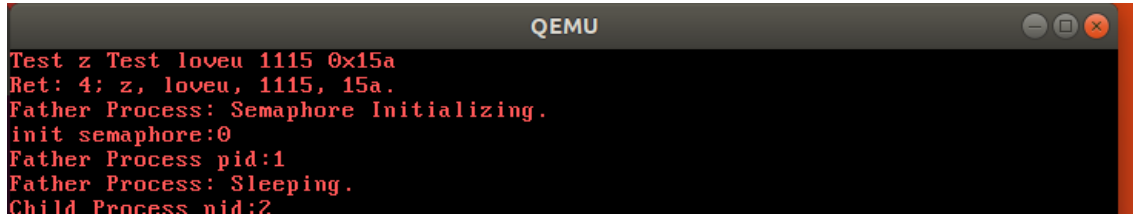
```
1 sf->eax = i;
```

☒ `dev[STD_IN].value > 0`，设备资源空闲，直接读取字符缓存,读取方式和阻塞进程唤醒后一致，不加赘述。

☒ `dev[STD_IN].value < 0`，本实验最多只有一个进程可以阻塞在设备上，读取失败。

```
1 sf->eax = -1;
```

- 运行截图



```
Test z Test loveu 1115 0x15a
Ret: 4; z, loveu, 1115, 15a.
Father Process: Semaphore Initializing.
init semaphore:0
Father Process pid:1
Father Process: Sleeping.
Child Process pid:2
```

## task2

- 实现下面四个函数

### ✓ syscallSemInit: 注册PV信号量

```
1 //syscall.c
2 *sem = syscall(SYS_SEM, SEM_INIT, value, 0, 0, 0);
```

可以看出系统调用传入一个参数：信号量的初值value，并存在一个返回值：返回信号量在内核中的序号。

- 遍历内核的**信号量数组**，查找是否存在未被使用的信号量。

```
1 for (i = 0; i < MAX_SEM_NUM; i++)
2 {
3     if (sem[i].state == 0) //在当前状态下信号量未被注册
4         break;
5 }
```

- 查到未使用的信号量，注意用传入的value来初始化信号量的处置并返回序号。

```
1 if (i < MAX_SEM_NUM)
2 {
3     sem[i].state = 1; // 0: not in use; 1: in use;
4     sem[i].value = sf->edx;
5     sf->eax = i;
6     return;
7 }
```

- 所有内核中的信号量都已经被注册，返回负一。

```
1 sf->eax = -1;
```

### ✓ syscallSemWait: 实现信号量P操作

```
1 return syscall(SYS_SEM, SEM_WAIT, *sem, 0, 0, 0);
```

可以看出只传入了信号量的序号来访问到对应的信号量。

首先在内核中要检验传入的参数是否有效。

```
1 if (i < 0 || i >= MAX_SEM_NUM)
2 {
3     pcb[current].regs.eax = -1;
4     return;
5 }
```

如果合法，和设备管理的处理方式基本一致，不再赘述，区别是这里信号量可以代表更多资源，也可以有不止一个进程阻塞在信号量上。

✅ **syscallSemPost**: 实现信号量V操作。

```
1 return syscall(SYS_SEM, SEM_POST, *sem, 0, 0, 0);
```

可以看出只传入了信号量的序号来访问到对应的信号量。

在内核中检测参数是否有效：

```
1 if (i < 0 || i >= MAX_SEM_NUM)
2 {
3     pcb[current].regs.eax = -1;
4     return;
5 }
```

和设备管理的思路几乎一致。

注意sem[i].value <= 0，唤醒在信号量上阻塞的一个进程，可以通过将进程的状态修改为就绪态触发时钟中断实现。

✅ **syscallSemDestory**: 实现信号量注销操作。

先检测参数是否合法。

在框架代码中，信号量以**固定大小数组**的形式给出，归还信号量只需要将信号量的**状态更改为未使用**即可。

```
1 sem[i].state = 0
```

但是特别要注意的是，如果有**进程仍旧阻塞在信号量上**是不允许注销信号量的。

```
1 if (pt != (ProcessTable*)&(sem[i].pcb))//exist precess blocked by the
   semaphore
2 {
3     sf->eax = -1;
4     return;
5 }
```

- 运行截图

```

init semaphore:0
Father Process pid:1
Father Process: Sleeping.
Child Process pid:2
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 Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.

```

### task3

- 完成app里面的下列问题，在报告里放上运行截图（注意在写其中一个问题时，把别的代码注释掉）。

#### ☒ 哲学家就餐问题

- 执行结果

```

Philosopher 5: pick down knife 0
Philosopher 1: pick up knife 0
Philosopher 1: eat with 0 and 1
Philosopher 2: think
Philosopher 3: eat with 2 and 3
Philosopher 5: pick down knife 4
Philosopher 1: pick down knife 1
Philosopher 2: pick up knife 1
Philosopher 3: pick down knife 3
Philosopher 4: pick up knife 3
Philosopher 5: think
Philosopher 1: pick down knife 0
Philosopher 3: pick down knife 2
Philosopher 4: pick up knife 4
Philosopher 5: pick up knife 0
Philosopher 2: pick up knife 2
Philosopher 1: think
Philosopher 2: eat with 1 and 2
Philosopher 3: think
Philosopher 4: eat with 3 and 4
Philosopher 2: pick down knife 1
Philosopher 4: pick down knife 3
Philosopher 1: pick up knife 1
Philosopher 3: pick up knife 3

```

#### ☒ 生产者消费者问题

- 执行结果
- （看起来这么有规律是因为我修改了sleep的时间来检验consumer可以阻塞在信号量上，毕竟消费者太少了，如果不缩短时间的话根本消费不完）

```

Producer 1: producing...
Producer 2: producing...
Producer 3: producing...
Producer 4: producing...
Producer 1: produce
Producer 2: produce
Producer 3: produce
Producer 4: produce
Consumer consume...
Consumer waiting...
Consumer consume...
Consumer waiting...
Consumer consume...
Consumer waiting...
Consumer consume...
Consumer waiting...
Producer 1: producing...
Producer 2: producing...
Producer 3: producing...
Producer 4: producing...
Producer 1: produce
Producer 2: produce

```

#### ☑ 读者写者问题

- 执行结果
- (可以看出一旦读者占有了读写锁就会把写者挡在外面，等所有进程读完才会释放读写锁)

```

Write process: 1, waiting...
Write process obtain ReadWritelock
Writer 2: write
Write process: 2, waiting...
Write process obtain ReadWritelock
Writer 3: write
Write process: 3, waiting...
Readprocess obtain ReadWritelock
Reader 2: read, total 1 reader
Reader 3: read, total 2 reader
Reader 1: read, total 3 reader
Read process: 2, waiting...
Read process: 3, waiting...
release ReadWritelock
Read process: 1, waiting...
Write process obtain ReadWritelock
Writer 1: write
ZWWrite process: 1, waiting...
Write process obtain ReadWritelock
Writer 2: write
Write process: 2, waiting...
Write process obtain ReadWritelock
Writer 3: write
Write process: 3, waiting...

```

## 四、PART4: feeling

- 噯，不知道说什么，就谢谢zjgg的辛勤付出吧。

