

# 操作系统实验

---

Lab4:

格式化输入函数 scanf、信号量相关系统调用、  
用户程序实现非死锁、高并行的哲学家问题

---

院系：人工智能学院

姓名：石睿

学号：211300024

班级：操作系统-2023 春季学期

邮箱：[211300024@smail.nju.edu.cn](mailto:211300024@smail.nju.edu.cn)

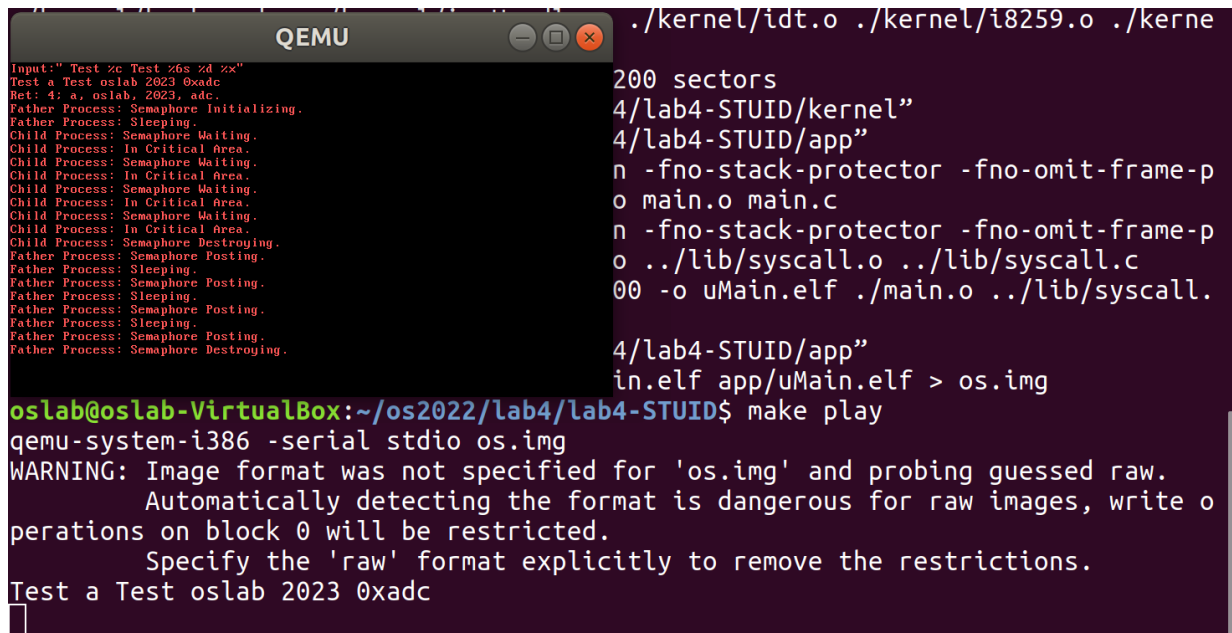
实验时间：2022.5.18

## 一、 实验进度

我已经完成了 Lab4 的所有内容。

## 二、 实验结果

结果如下图所示，具体实现过程在第三部分介绍。



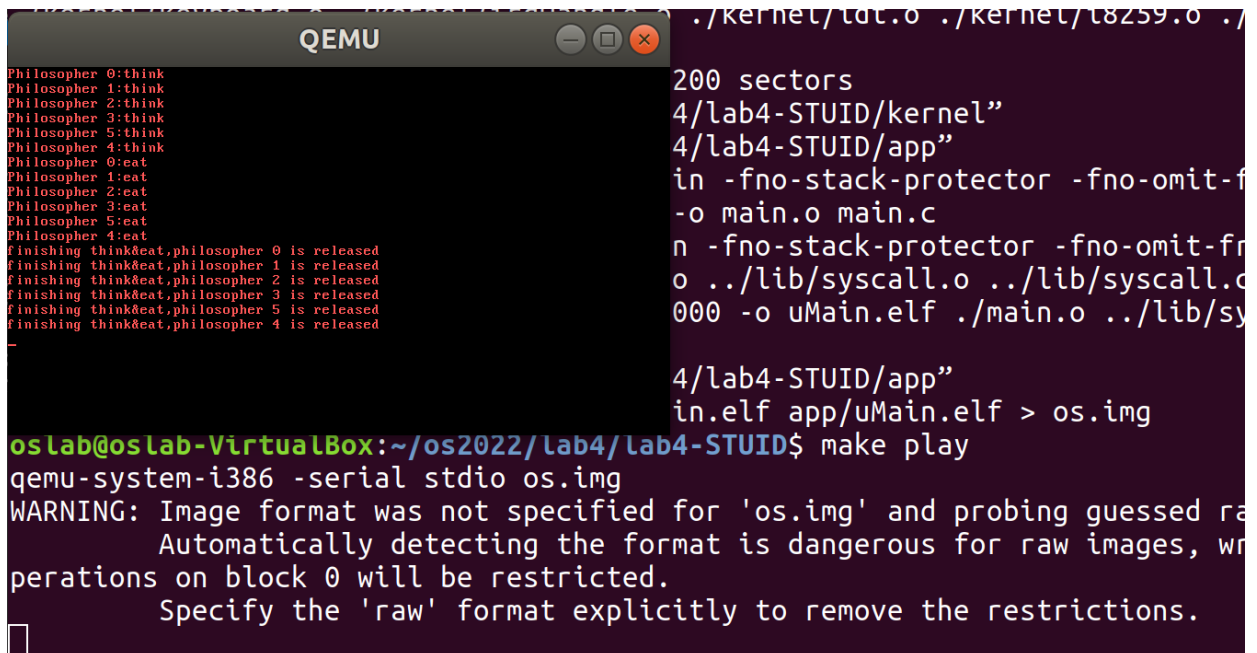
```
QEMU
Input: "Test xc Test x6s xd xx"
Test a Test oslab 2023 0xad
Ret: 4: a, oslab, 2023, adc.
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.

200 sectors
4/lab4-STUID/kernel"
4/lab4-STUID/app"
n -fno-stack-protector -fno-omit-frame-p
o main.o main.c
n -fno-stack-protector -fno-omit-frame-p
o ../lib/syscall.o ../lib/syscall.c
00 -o uMain.elf ./main.o ../lib/syscall.

4/lab4-STUID/app"
in.elf app/uMain.elf > os.img

oslab@oslab-VirtualBox:~/os2022/lab4/lab4-STUID$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
Test a Test oslab 2023 0xad
```

完成格式化输入函数相关中断处理例程后结果



```
QEMU
Philosopher 0:think
Philosopher 1:think
Philosopher 2:think
Philosopher 3:think
Philosopher 5:think
Philosopher 4:think
Philosopher 0:eat
Philosopher 1:eat
Philosopher 2:eat
Philosopher 3:eat
Philosopher 5:eat
Philosopher 4:eat
finishing think&eat,philosopher 0 is released
finishing think&eat,philosopher 1 is released
finishing think&eat,philosopher 2 is released
finishing think&eat,philosopher 3 is released
finishing think&eat,philosopher 5 is released
finishing think&eat,philosopher 4 is released

200 sectors
4/lab4-STUID/kernel"
4/lab4-STUID/app"
in -fno-stack-protector -fno-omit-fr
-o main.o main.c
n -fno-stack-protector -fno-omit-fr
o ../lib/syscall.o ../lib/syscall.c
000 -o uMain.elf ./main.o ../lib/sy

4/lab4-STUID/app"
in.elf app/uMain.elf > os.img

oslab@oslab-VirtualBox:~/os2022/lab4/lab4-STUID$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed ra
Automatically detecting the format is dangerous for raw images, wr
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

只跑一轮（for 循环 i=1）、哲学家初始化（now\_state 数组）为 Thinking 时的运行结果

### 三、 实验修改的代码位置

#### ① 格式化输入函数 scanf

Syscall.c 中有关用户程序可以使用的 scanf 已经实现（调用了许多系统调用），只需要完成中断处理程序分发函数 irqHandle.c 中对应处理用户键盘输入的中断处理例程 keyboardHandle 和 syscallReadStdIn 两个函数。

##### 1.1 实现 keyboardHandle 函数 = 写 keybuffer

本函数应完成的功能为[1]和[2]，以下对完成两个功能分别介绍：

[1]按下键盘中断后，把本字符所对应的 keycode 放到 keybuffer 之中。

按下键盘中断后，应该同时处理 qemu 的显示（显存）以及串口（serial）的显示，具体代码和 lab2 中一毛一样啦！需要处理：非法字符、退格符、回车符、正常字符的处理，如果合法，就放入 keybuffer 之中。放入 keybuffer 中的代码如下所示。

完成放入 keybuffer 之后，再滚屏（scrollscreen），并更新屏幕 updateCursor 即可。

```
data = 0 | (0x0c << 8);
pos = (80*displayRow+displayCol)*2;
asm volatile("movw %0, (%1)":"r"(data),"r"(pos+0xb8000));
```

[2]执行信号量 V()操作，唤醒 dev[STD\_IN]阻塞列表中的某一个进程。

使用内核初始化 initSem 中定义的 Device 信号量中的 dev[STD\_IN]，在键盘中断将输入存入缓冲区后再让用户程序读取，而非一直监听键盘中断进行输入浪费资源。

由规定可知，最多只能有一个进程被阻塞到 dev[STD\_IN]上，而如果有进程可以读 keybuffer，那么他应该可以把 keybuffer 中的内容完全读入！故此时 dev[STD\_IN].value 可以理解为（和传统信号量不太一样）：

1、当 value<0 的时候，表示阻塞在 dev[STD\_IN]标准输入任务上的进程数量，且

只能是-1。

2、当 value>0 的时候，表示当前在 keybuffer 中没有读入的字符个数。如果当前有进程从 dev[STD\_IN]阻塞列表中唤醒，那么就应该全部读完，即让 dev[STD\_IN]=0!

完成对 dev[STD\_IN].value 的更新后，就应该对唤醒的进程状态 (processtable) 的状态进行更新，并最后产生时间中断，由调度器完成下一次的进程调度。

具体代码实现如下。

```
// 【功能2: V()操作! 1、唤醒阻塞在dev[STD_IN]上的一个进程 2、创建资源】最多只能有一个进程被阻塞在dev[STD_IN]上
//此时有buffer的输入，有资源可以读入啦!
if (dev[STD_IN].state == 1 && dev[STD_IN].value < 0) { // with process blocked,此时需要唤醒他
    // TODO: deal with blocked situation
    // 【先把他从阻塞列表中拿出来】
    ProcessTable* pt;
    pt = (ProcessTable*)((uint32_t)(dev[STD_IN].pcb.prev - (uint32_t)&(((ProcessTable*)0)->blocked));
    dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
    (dev[STD_IN].pcb.prev)->next = &(dev[STD_IN].pcb);
    // 【再对他的信息进行修改】
    pt->state=STATE_RUNNABLE;
    //pt->timecount=MAX_TIME_COUNT;这两行最好别加，为了保证唤醒后的运行逻辑不发生改，原来时间片剩多长时间
    //pt->sleeptime=0;-----加这两行也不会错，只是调度会和期望的不同
    dev[STD_IN].value = 1; // 【因为最多只能有一个线程被阻塞，唤醒了之后全部可以读! 》》此时有一个字符可以由本读入
    asm volatile("int $0x20"); //符合逻辑，此时有一个进程被唤醒，发时间中断，看调度器会不会调度他! -----不加应该也没
    return;
}
```

## 1.2 实现 syscallReadStdIn 函数 = 读 keybuffer

本函数应该完成的功能为[1]和[2]，以下分别进行介绍：

首先，注意到规定：“如果多个进程想读，那么后来的进程直接返回-1”。并且结合“最多只能有一个进程被阻塞在 dev[STD\_IN]上”，故不把新进程加入阻塞列表中，直接 pcb[current].regs.eax = -1;就可以啦!

[1]如果 dev[STD\_IN].value=0（没有多余字符可读&没有其他阻塞进程），当前进程阻塞。

基本和 V()的处理思路相同：加入到 dev[STD\_IN]阻塞列表中，更新当前进程 pcb[current]的状态，更新 dev[STD\_IN].value 的值（自减其实就相当于把 value 的值直接赋值为-1 啦!）。

值得注意的是，此时调用 scanf 的返回值应该为 0（当前进程一个字符也读不出

来)，最后调用时间中断，由调度器完成调度。代码实现如下

```
else if(dev[STD_IN].state==1 && dev[STD_IN].value==0){/*一个进程想读&& 而且没有资源。加入阻塞列表+修改信息
    pcb[current].blocked.next = dev[STD_IN].pcb.next;
    pcb[current].blocked.prev = &(dev[STD_IN].pcb);
    dev[STD_IN].pcb.next = &(pcb[current].blocked);
    (pcb[current].blocked.next->prev = &(pcb[current].blocked));

    pcb[current].state = STATE_BLOCKED;
    pcb[current].sleepTime = 0;
    pcb[current].regs.eax = 0; /*XXX: 一个进程&没有资源，返回0（一个都读不出来）
    dev[STD_IN].value--;
    asm volatile("int $0x20");
    return ;
}
```

## [2]进程被唤醒，读取 keybuffer 中所有的数据

此时只需要额外留意 scanf 的返回值，以及当前被唤醒的进程可以把 keybuffer 上面的字符全部读完，应显示赋值为 0！（自减是不对的呢！）

```
int sel = sf->ds;
char *str = (char *)sf->edx;
int size = sf->ebx;
int i = 0;
char character = 0;
asm volatile("movw %0, %%es" ::"m"(sel));
for (i = 0; i < size && bufferHead != bufferTail; i++){
    character = keyBuffer[bufferHead];
    bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
    asm volatile("movb %0, %%es:(%1)" ::"r"(character), "r"(str + i));
}
asm volatile("movb $0x00, %%es:(%0)" ::"r"(str + i));
pcb[current].regs.eax = i; /*XXX: 没有进程阻塞&有资源，返回读入的个数！
dev[STD_IN].value=0;
return ;
```

## ② 信号量相关系统调用

注意到 app/main.c 中 printf 了 "In Critical Area"，所以以下所有实现之前均进行了关中断操作，并在完成状态更新后开中断。

有以下值得注意的要点：

- 1、 syscall 中参数存放的位置 eax(系统调用号) ecx（参数 1） edx（参数 2），参数存放的位置是 StackFrame 在 dolrq.S 中压栈后形成的数据结构中的位置决定的。
- 2、 需要考虑输入不合法的情况，如到 MAX\_SEM\_NUM 后，没有位置给新信号量初始

化、信号量数组的下标非法（小于 0，大于最大值,,,）

- 3、 同第一步的实现，需要在 P 操作 `syscallSemWait` 和 V 操作 `syscallSemPost` 的最后引发时钟中断，由调度器决定接下来该谁执行啦！

其他实现依据实验手册逐步完成即可啦！

### ③ 哲学家就餐问题

为了保证哲学家问题不出现死锁，且保证较高的并行度，并尽可能让最多的哲学家拿到筷子进行吃饭，对手册上的提到的可以不死锁的算法进行更改

```
int state[5]; // keep track of every philosopher's state
               // {HUNGRY, EATING, THINKING}
sem_t s[5]; // one semaphore per philosopher -> 0
sem_t mutex = 1; // mutex lock

void get_fork(i) {
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}

void put_fork(i) {
    P(mutex);
    state[i] = THINKING;
    test(left);
    test(right);
    V(mutex);
}

void test(i) {
    // whether the two forks can be acquired
    if (state[i] == HUNGRY && state[left] != EATING &&
        state[right] != EATING) {
        state[i] = EATING;
        V(s[i]);
    }
}
```

思路如下所述：

- 1、想吃饭的时候：只有在这哲学家同时具有左右两支筷子的时候才会拿起筷子并吃饭。
- 2、吃完饭的时候：放下筷子，同时检测左右两个相邻的哲学家有没有吃饭的需求，检测他们是否可以拿起筷子吃饭（左右两筷子都空闲）。

在这两个过程中，都用互斥锁放入临界区之中，只需要用信号量把初始值 value 设置成 1。

在实现的过程中值得一提的是，创建哲学家的过程中需要确认当前哲学家的编号。因为在 `interleave` 的过程中，哲学家是交织执行的，只知道哲学家的 pid，但不知道哲学家的具体编号。可以通过 `fork` 函数的两次返回值指示当前哲学家的编号！`fork` 第一次在子进程中返回 0，第二次在父进程



中返回子进程的 pid，所以可以通过 fork 的返回值是否为 0，判断当前给哲学家分配的编号是几！

```
// 【step2: 创建五个哲学家，确定当前哲学家的编号cur_ph】
int cur_ph; // XXX: 指示当前的进程是谁,不是pid!!! 而是五个哲学家中的编号(0-4)!
pid_t new_ph; // XXX: 指示新的进程是谁，是pid!!!
for(cur_ph=0;cur_ph<5;cur_ph++){
    new_ph = fork();//fork返回两次，父进程返回子的pid，子进程返回0!
    if(new_ph==0)//子进程返回，此时cur_ph就指示当前子进程中哲学家的编号是什么了!
        break;
    else if(cur_ph<0){
        printf("something wrong with creating new philosophers\n");
        exit();
    }
}
```

可以通过在在代码标记 step3 的位置修改给定 for 循环的轮次、修改初始化 now\_state 数组、（哲学家的初始状态）、修改 forks 数组（哲学家当前可不可吃饭）来增加测试用例。

Tip: 提交代码的时候，已把哲学家问题在 app.main.c 中屏蔽掉啦。如要测试，还请把 lab4.1 和 lab4.2 的内容屏蔽，并把对 lab4.3 的屏蔽内容删除。