# Lab4 实验报告


# 231880101 孙俊晖

# 邮箱：2033282932@qq.com


## 一、实验完成情况

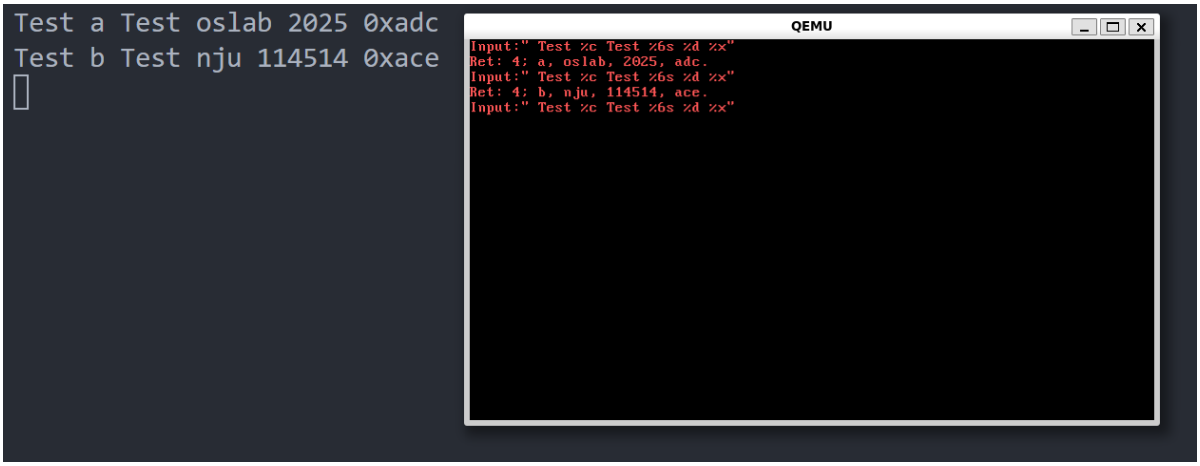
### 1. 实验进度

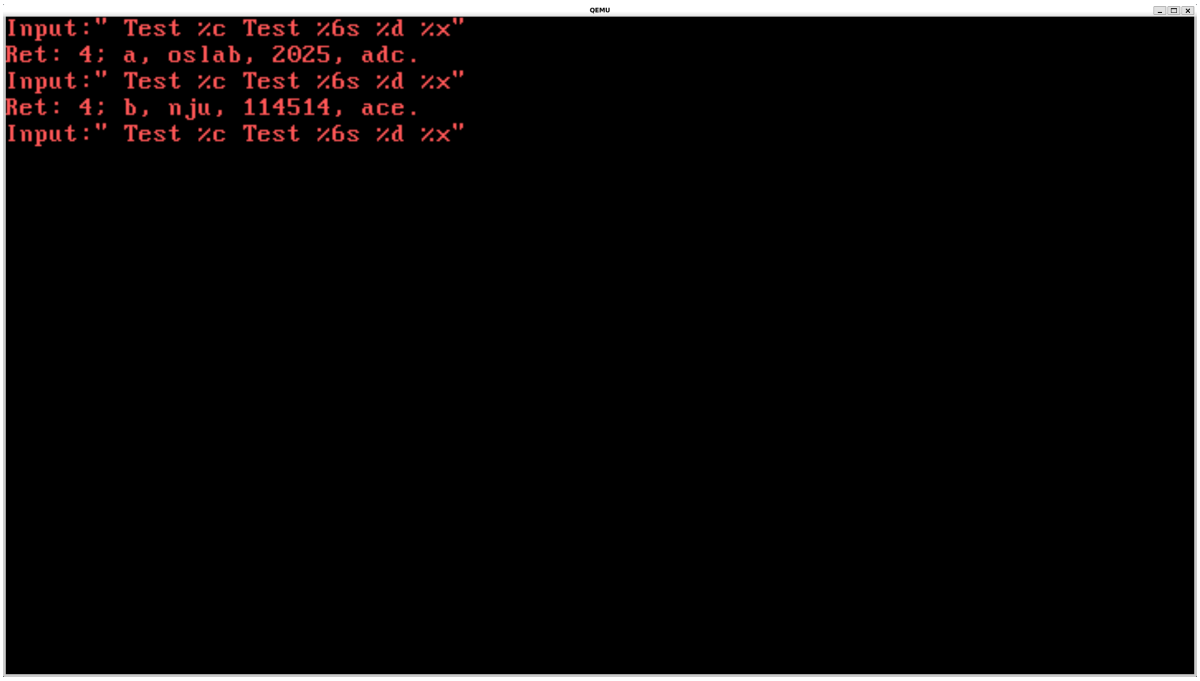
我完成了除选做部分3.3.2以外的所有内容。

### 2. 实验结果

#### （1）实现格式化输入函数

我在实现格式化输入函数后，用实验文档1.1部分给出的测试代码进行测试，结果如下：




首先，我测试了实验文档提供的测试样例：

在输入 Test a Test oslab 2025 0xadc 后，屏幕上输出为 Ret: 4; a, oslab, 2025, adc.，与预计输出一致；

此外，我还测试了自己编写的样例：

在输入 Test b Test nju 114514 0xace 后，屏幕上输出为 Ret: 4; b, nju, 114514, ace.，同样与预计输出一致。

### （2）实现信号量

我在实现 SEM_INIT、SEM_POST、SEM_WAIT、SEM_DESTROY 系统调用后，用实验文档1.2部分给出的代码进行测试，结果如下：

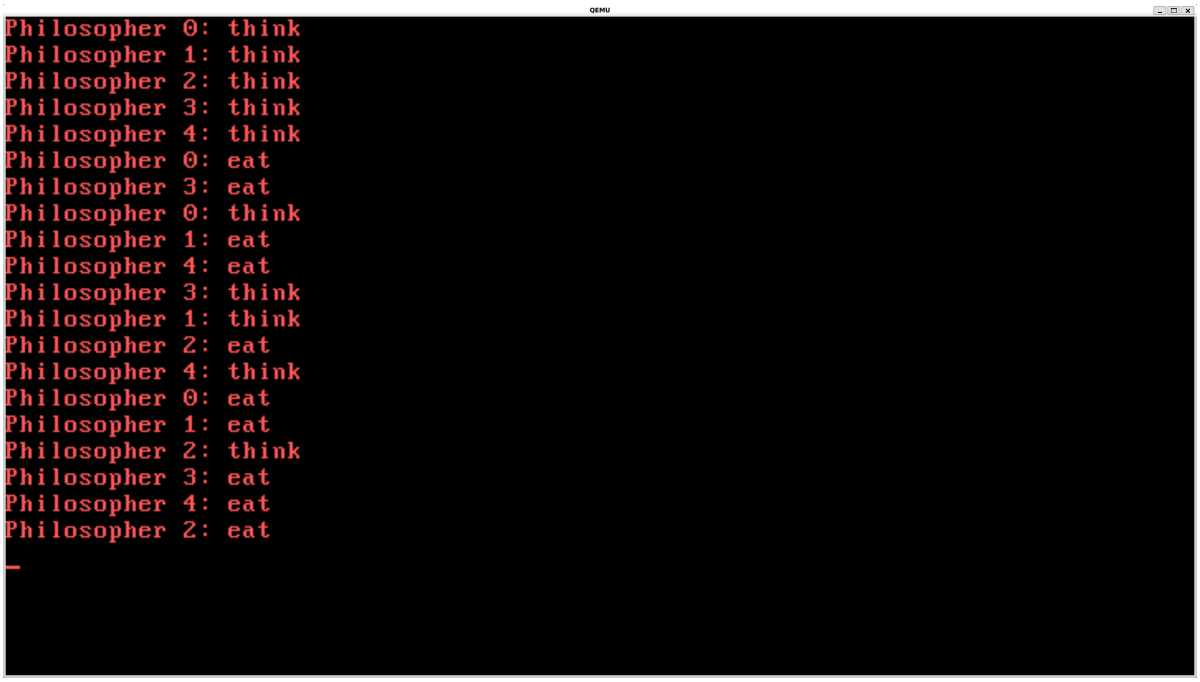

对测试代码的输出结果的分析如下：

1. **信号量初始化成功**：父进程输出 Semaphore Initializing，未提示失败，表明 sem_init 返回值为 0。
2. **进程同步逻辑正确**：
   （1）子进程前两次直接进入临界区（无父进程输出干扰）。
   （2）父进程每次调用 sem_post 后，子进程从阻塞中恢复，后续两次进入临界区。
   （3）最终父子进程均调用 sem_destroy，销毁信号量。
3. **无死锁与资源泄漏**：
   （1）子进程4次进入临界区后正常退出，父进程4次发布信号量后销毁信号量，表明无死锁。
   （2）信号量最终被销毁，验证了 sem_destroy 的正确性。

这说明测试代码符合预期，SEM_INIT、SEM_POST、SEM_WAIT、SEM_DESTROY 系统调用均正确实现。

### （3）实现哲学家就餐问题

我实现了哲学家就餐问题，使得5个哲学家同时运行，保证每个哲学家能正常地进行思考和就餐，结果如下：




从输出结果来看，不同ID的哲学家交替打印 eat，说明所有哲学家能交替进餐，信号量操作逻辑正确。并且，由输出结果可知，并没有出现死锁现象，说明我基本上成功实现了哲学家就餐问题。

# 二、实验过程

## 1. 实现格式化输入函数

由于 lab4/lib/syscall.c 中已经完成了 scanf 函数，这部分只需要完成对应的中断处理例程即可。具体来说，需要完成 lab4/kernel/kernel/irqHandle.c 中的 keyboardHandle 函数和 syscallReadStdIn 函数。

### （1）完成 keyboardHandle 函数

首先读取 keyCode， 如果读取到的 keyCode 是合法的（keyCode $\neq 0$）就放入到 keyBuffer 中。

```
uint32_t keyCode = getKeyCode();
if (keyCode == 0) // illegal keyCode
    return;
keyBuffer[bufferTail] = keyCode;
bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;
```

然后，如果有阻塞进程，就唤醒阻塞在 dev[STD_IN] 上的一个进程。这里用 dev[STD_IN].value 表示阻塞进程数量，dev[STD_IN].value = -1 表示共有1个阻塞进程，dev[STD_IN].value = -2 表示共有2个阻塞进程，以此类推，dev[STD_IN].value = -n 表示有n个阻塞进程。因此，存在阻塞进程当且仅当 dev[STD_IN].value < 0。实验文档中已经给出了从信号量 i 上阻塞的进程列表取出一个进程的代码，因此这里取出一个被阻塞的进程的代码直接参照实验文档中给出的代码就行了。唤醒进程的代码也很容易，只需要将状态设为可运行状态即可。最后，由于已经从阻塞队列移除了唤醒的进程，这里需要更新阻塞进程的数量，即更新 dev[STD_IN].value 的值。

```
if (dev[STD_IN].value < 0) { // with process blocked
    // TODO: deal with blocked situation
    // 取出第一个被阻塞的进程
    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);
    // 唤醒进程
    pt->state = STATE_RUNNABLE;
    dev[STD_IN].value++;
}
```

### （2）完成 syscallReadStdIn 函数

syscallReadStdIn 函数首先检查标准输入设备是否有可用数据：若 dev[STD_IN].value == 0 表明输入缓冲区为空，当前进程会被加入设备阻塞队列并置为 STATE_BLOCKED 状态，通过触发调度 int $0x20 主动让出CPU等待输入；当存在可用数据时，函数通过段寄存器操作将用户提供的缓冲区地址映射到内核空间，随后循环从keyBuffer 读取字符数据，每次读取后更新缓冲区指针并调用 putChar() 实现控制台回显，最终通过内联汇编将字符写入用户空间缓冲区，循环结束后在缓冲区末尾添加空字符作为字符串终止符，最后将实际读取的字节数存入eax 寄存器作为系统调用返回值。整个过程实现了从内核缓冲区到用户空间的受控数据传输，并通过进程阻塞机制避免了忙等状态。其中，阻塞当前进程的代码可以参考实验文档中给出的将 current 线程加到信号量 i 的阻塞列表的代码。同样地，实验文档中也给出了将读取的字符传到用户进程的代码，这样读 keyBuffer 中的所有数据的这部分代码也不难实现。

```
// 如果dev[STD_IN].value == 0，将当前进程阻塞在dev[STD_IN]上
if(dev[STD_IN].value == 0)
{
    // 阻塞当前进程
    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 = -1;
    dev[STD_IN].value--;
    // 触发调度
    asm volatile("int $0x20");
    // 读取键盘缓冲区数据
    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));
    while (i < size && bufferHead != bufferTail) {
        character = getChar(keyBuffer[bufferHead]);
        bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
        putChar(character);
        if(character != 0)
        {
            asm volatile("movb %0, %%es:(%1)"::"r"(character), "r"(str +
i));
            i++;
```

```
        }
    }
    asm volatile("movb $0, %%es:(%0)"::"r"(str + i)); // 添加字符串结束符
    pcb[current].regs.eax = i; // 返回读取的字节数
    return;
}
// 如果已有进程阻塞,直接返回失败
else if(dev[STD_IN].value < 0)
{
    pcb[current].regs.eax = -1;
    return;
}
```

## 2. 实现信号量

这部分需要完成的是lab4/kernel/kernel/irqHandle.c 中的 4 个函数：syscallSemInit、syscallSemWait、syscallSemPost 和 syscallSemDestroy 以实现 SEM_INIT、SEM_POST、SEM_WAIT、SEM_DESTROY 系统调用。

### （1）完成 syscallSemInit 函数

该函数实现信号量的初始化操作。它遍历全局信号量数组，找到第一个未被使用的信号量（state == 0），将其状态标记为已初始化（state = 1），并通过参数设置信号量的初始计数值。同时初始化信号量关联的进程阻塞队列（双向循环链表），最终返回新信号量的索引。若无空闲信号量，则返回错误码 -1。

```
void syscallSemInit(struct StackFrame *sf) {
    // TODO: complete `SemInit`
    for(int i = 0; i < MAX_SEM_NUM; i++)
    {
        if(sem[i].state == 0)
        {
            sem[i].state = 1;
            sem[i].value = (int32_t)sf->edx;
            sem[i].pcb.next = &(sem[i].pcb);
            sem[i].pcb.prev = &(sem[i].pcb);
            pcb[current].regs.eax = i;
            return;
        }
    }
    pcb[current].regs.eax = -1;
    return;
}
```

### （2）完成 syscallSemWait 函数

该函数执行信号量的 P 操作（等待/获取资源）。首先验证信号量有效性，若合法则将信号量计数值减 1。若减 1 后计数值小于 0，表明资源不足，当前进程会被加入信号量的阻塞队列，并修改进程状态为 STATE_BLOCKED，最后通过触发调度（int $0x20）主动让出 CPU。操作结果通过 eax 寄存器返回（成功返回 0，无效信号量返回 -1）。

```
void syscallSemWait(struct StackFrame *sf) {
    // TODO: complete `SemWait` and note that you need to consider some
special situations
```

```
    // 用户传入的信号量索引
    int i = (int)sf->edx;
    // 操作失败，返回-1
    if(i < 0 || i >= MAX_SEM_NUM || sem[i].state == 0)
    {
        pcb[current].regs.eax = -1;
        return;
    }
    // P操作：信号量值减1
    sem[i].value--;
    // 若value取值小于0,则阻塞自身
    if (sem[i].value < 0) {
        // 将当前进程加入信号量的阻塞队列
        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);
        pcb[current].state = STATE_BLOCKED;
        pcb[current].sleepTime = -1;
        // 触发调度
        asm volatile("int $0x20");
    }
    // 操作成功，返回0
    pcb[current].regs.eax = 0;
    return;
}
```

### （3）完成 syscallSemPost 函数

该函数执行信号量的 V 操作（释放/增加资源）。验证信号量有效性后，将信号量计数值加 1。若加 1 后计数值仍 不大于 0，说明有进程阻塞在该信号量上，此时从阻塞队列头部唤醒一个进程，将其状态设置为 STATE_RUNNABLE。操作结果通过 eax 寄存器返回（成功返回 0，无效信号量返回 -1）。

```
void syscallSemPost(struct StackFrame *sf) {
    int i = (int)sf->edx;
    ProcessTable *pt = NULL;
    if (i < 0 || i >= MAX_SEM_NUM || sem[i].state == 0) {
        pcb[current].regs.eax = -1;
        return;
    }
    // TODO: complete other situations
    // V操作：信号量值加1
    sem[i].value++;
    // 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)
    if(sem[i].value <= 0)
    {
        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);
        pt->state = STATE_RUNNABLE;
        pt->sleepTime = 0;
    }
    // 操作成功，返回0
    pcb[current].regs.eax = 0;
```

```
    return;
}
```

### （4）完成 syscallSemDestory 函数

该函数用于安全销毁信号量。它会严格检查信号量状态：必须已初始化且阻塞队列为空。若验证通过，则将信号量状态重置为未初始化（state = 0），清空计数值，并通过触发调度确保潜在等待进程被唤醒。若信号量无效或仍有进程阻塞，则返回错误码 -1；否则返回 0 表示销毁成功。

```
void syscallSemDestroy(struct StackFrame *sf) {
    // TODO: complete `SemDestroy`
    int i = sf->edx;
    if(i < 0 || i >= MAX_SEM_NUM || sem[i].state == 0 || sem[i].pcb.next != &sem[i].pcb)
    {
        pcb[current].regs.eax = -1;
        return;
    }
    sem[i].state = 0;
    sem[i].value = 0;
    asm volatile("int $0x20");
    pcb[current].regs.eax = 0;
    return;
}
```

## 3. 实现哲学家就餐问题

首先，我实现了 getpid 系统调用，用来返回当前进程的 pid。

```
#define SYS_PID 7
// 需要修改lab4/kernel/kernel/irqHandle.c中的syscallHandle函数
void syscallHandle(struct StackFrame *sf) {
    switch(sf->eax) { // syscall number
        case SYS_WRITE:
            syscallWrite(sf);
            break; // for SYS_WRITE
        ......
        case SYS_PID:
            syscallPid(sf);
            break; // for SYS_PID
        default:break;
    }
}
// 该函数定义在lab4/kernel/kernel/irqHandle.c中
void syscallPid(struct StackFrame *sf)
{
    pcb[current].regs.eax = current;
    return;
}
// 该函数定义在lab4/lib/syscall.c中
pid_t getpid()
{
    return syscall(SYS_PID, 0, 0, 0, 0, 0);
}
```

接下来就是哲学家就餐问题的具体实现。我用5个信号量 forks[5] 模拟叉子资源，通过信号量机制实现了5个并发进程对共享资源的互斥访问，采用实验文档中提示的方案3——奇偶哲学家反向拿叉策略（偶数号哲学家按左→右顺序获取叉子，奇数号哲学家按右→左顺序）有效打破了循环等待条件，从而预防了死锁发生。实验中通过fork()创建4个子进程配合父进程模拟5位哲学家，利用PID偏移量确定哲学家ID，每个进程执行2次"思考-进餐"循环并严格通过sem_wait()/sem_post()操作管理叉子资源，最终输出结果显示哲学家交替进餐且无阻塞现象，验证了方案的正确性。

```c
#include "lib.h"
#include "types.h"

int uEntry(void) {
    // For lab4.3
    // TODO: You need to design and test the philosopher problem.
    // Note that you can create your own functions.
    // Requirements are demonstrated in the guide.
    // For Lab4.3.1
    int ret = 0;
    sem_t forks[5];
    // 信号量初始化
    for(int i = 0; i < 5; i++)
    {
        sem_init(&forks[i], 1);
    }
    // 创建四个子进程，加上父进程共5个进程，对应五位哲学家
    for(int i = 0; i < 4; i++)
    {
        if(ret == 0)
        {
            ret = fork();
        }
        // 父进程跳出循环
        else if(ret > 0)
        {
            break;
        }
    }
    // 获取进程ID
    int pid = getpid();
    // 计算哲学家ID
    int id = pid - 1;
    // 每个哲学家执行2次完整的"思考-进餐"循环
    for(int i = 0; i < 2; i++)
    {
        printf("Philosopher %d: think\n", id);
        sleep(128);
        if(id % 2 == 0)
        {
            sem_wait(&forks[id]);
            sem_wait(&forks[(id + 1) % 5]);
        }
        else
        {
            sem_wait(&forks[(id + 1) % 5]);
            sem_wait(&forks[id]);
        }
```

```
            // 进餐(拿起叉子)
            printf("Philosopher %d: eat\n", id);
            sleep(128);
            // 放下叉子
            sem_post(&forks[id]);
            sem_post(&forks[(id + 1) % 5]);
        }
        // 如果是子进程，直接退出
        if(id != 0)
        {
            exit();
        }
        // 销毁所有信号量
        for(int i = 0; i < 5; i++)
        {
            sem_destroy(&forks[i]);
        }
        exit();
        return 0;
}
```

# 三、实验中遇到的问题

## 1. 进程ID映射混乱

在实现哲学家就餐问题时，我初始使用 id = getpid() % 5 计算哲学家编号，导致子进程ID与父进程产生模运算冲突。后来我改用 id = pid - 1实现严格线性映射，确保0号进程为父进程，1-5号进程对应哲学家0-4，解决了进程身份错位问题。

## 2.资源泄漏隐患

在实现哲学家就餐问题时，初始版本未在子进程退出前释放信号量，导致父进程销毁信号量时可能访问已释放内存。后来我通过在子进程退出前增加 exit() 调用，确保信号量由父进程统一销毁，规避了资源重复释放问题。

# 四、实验总结

在本次实验中，我成功实现了格式化输入、信号量机制及哲学家就餐问题，了解了基于信号量的进程同步机制。这次实验不仅强化了我对操作系统同步机制的理论认知，更使我获得了将经典问题转化为实际代码的系统化工程能力，为后续学习更复杂的并发模型奠定了坚实基础。