lab4实验报告

陈德丹 221220159 邮箱：221220159@smail.nju.edu.cn

实验进度

完成所有实验内容+实现选做哲学家问题

实验过程

3.1. 实现格式化输入函数

keyboradHandle：
获取键盘扫描码并存入 keyBuffer 中，唤醒阻塞在 dev[STD_IN] 上的一个进程，将其切换为运行态
syscallReadStdIn：
如果 dev[STD_IN].value == 0 ，将当前进程阻塞在 dev[STD_IN] 上，调用时钟中断进行进程切换。之后进程被唤醒，读 keyBuffer 中的所有数据，并通过 str 将读取的字符传到用户进程（可以参考lab2中的代码）

3.2. 实现信号量

syscallSemInit：
遍历 sem[MAX_SEM_NUM] 找到一个未使用的信号量，用 sf->edx 初始化 value，返回该信号量的下标（若初始化失败则返回-1）
syscallSemWait：
将要申请资源的信号量的 value 减一，如果 value < 0 则阻塞当前进程
syscallSemPost：
将要释放资源的信号量的 value 加一，如果 value <= 0 则唤醒信号量等待队列中的一个进程
syscallSemDestroy：
销毁当前信号量，将state设为0，注意要重置pcb

3.3. 解决进程同步问题

3.3.1. 生产者-消费者问题

设立三个信号量 empty = 5, full = 0, mutex = 1，创建4个子进程作为生产者，代码如下：

```c
i = 0;
int id = 0;
sem_t empty, full, mutex;
sem_init(&empty, 5);
sem_init(&full, 0);
sem_init(&mutex, 1);
for (i = 0; i < 4; i++) {
        if (id == 0) id = fork();
        else if (id > 0) break;
}
while (1) {
        if (id == 0){
                sem_wait(&full);
                sem_wait(&mutex);
                printf("Consumer : consume\n");
                sleep(128);
                sem_post(&mutex);
                sem_post(&empty);
        }
        else{
                sem_wait(&empty);
                sem_wait(&mutex);
                printf("Producer %d: produce\n", id - 1);
                sleep(128);
                sem_post(&mutex);
                sem_post(&full);
        }
}
```

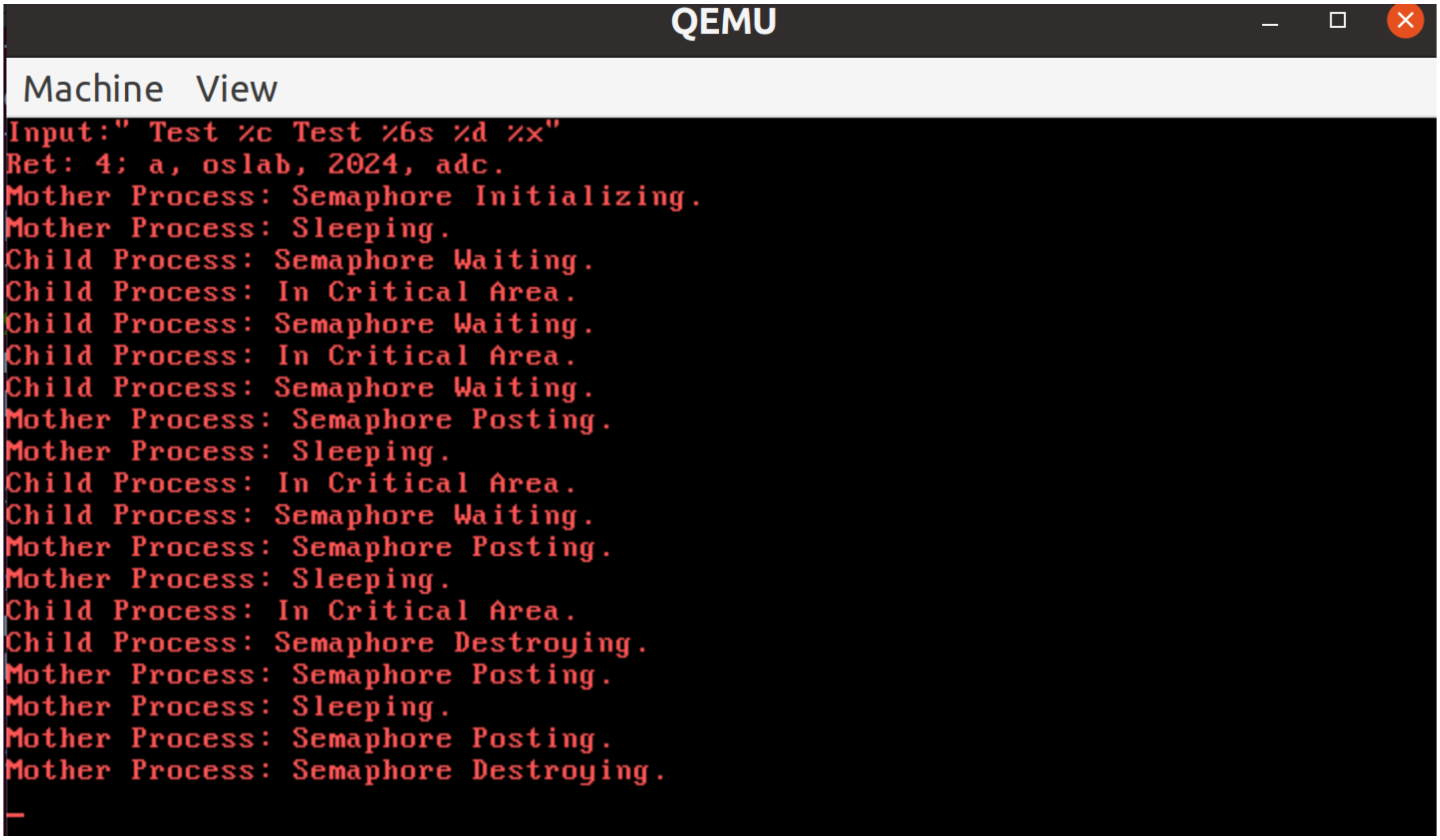
3.3.2. 哲学家问题

双数号哲学家先拿左边的叉子，再拿右边的叉子；单数号哲学家先拿右边的叉子，再拿左边的叉子。没有死锁，可以实现多人同时就餐。

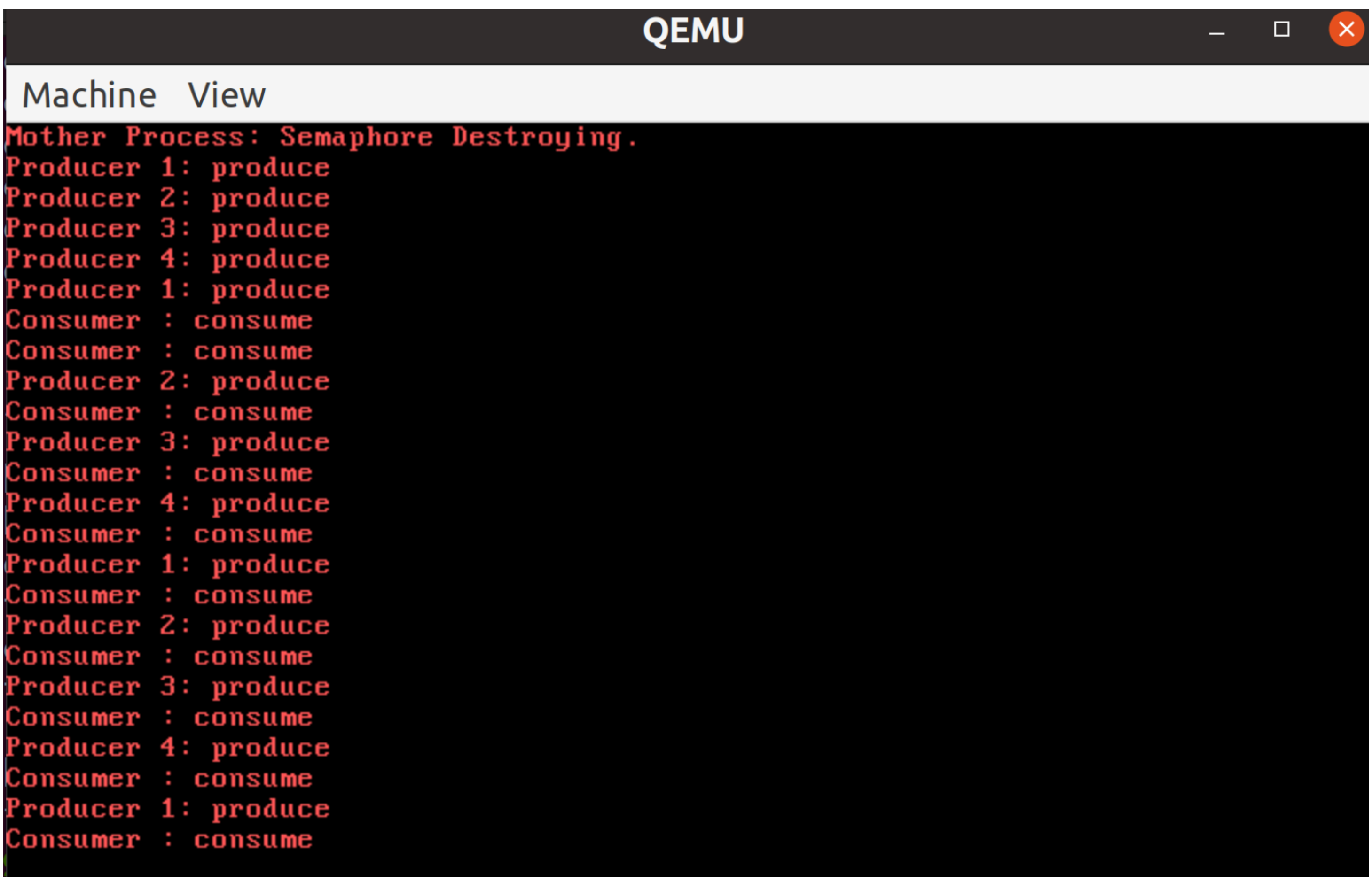
```c
int id = 0;
sem_t forks[5], mutex;
sem_init(&mutex, 1);
for (int i = 0; i < 4; i++) sem_init(&forks[i], 1);
for (int i = 0; i < 4; i++) {
        if (id == 0) id = fork();
        else if (id > 0) break;
}
if (id > 0) id -= 1;
while (1) {
        printf("Philosopher %d: think\n", id);
        sleep(128);
        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]);
        }
        printf("Philosopher %d: eat\n", id);
        sleep(128);
        sem_post(&forks[id]);
        sem_post(&forks[(id + 1) % 5]);
}
```

实验结果

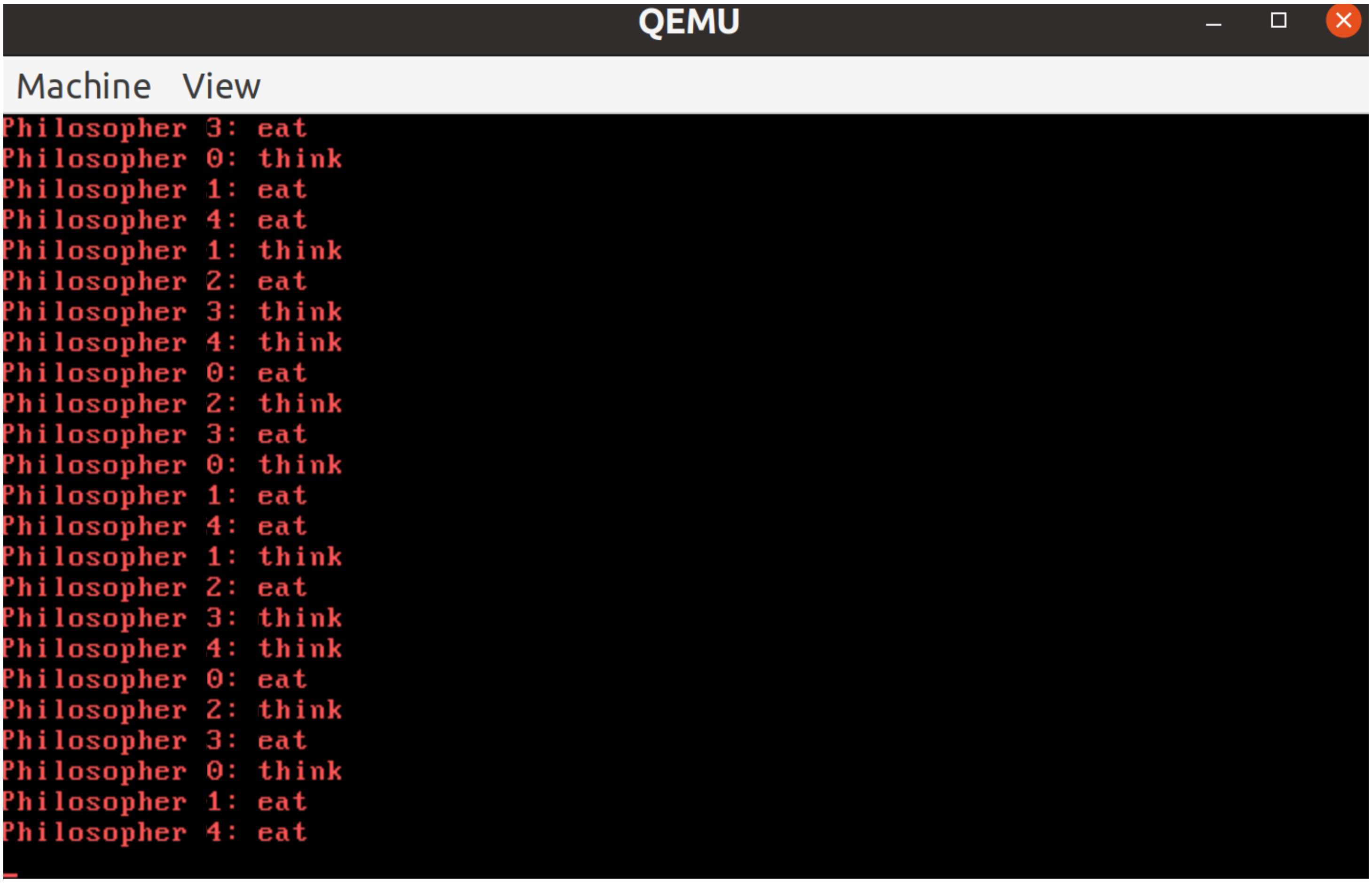
3.1 & 3.2



3.3.1 生产者-消费者问题



3.3.2 哲学家问题



其他

复习了一下系统调用的整个过程

主程序调用了sem_init函数

```
ret = sem_init(&sem, 2);
```

sem_init中调用了syscall函数

```
int sem_init(sem_t *sem, uint32_t value) {
        *sem = syscall(SYS_SEM, SEM_INIT, value, 0, 0, 0);
        if (*sem != -1)
                return 0;
        else
                return -1;
}
```

在syscall函数中，将eax等寄存器压栈，调用int $0x80指令

```
int32_t syscall(int num, uint32_t a1,uint32_t a2,
                uint32_t a3, uint32_t a4, uint32_t a5)
{
        int32_t ret = 0;
        uint32_t eax, ecx, edx, ebx, esi, edi;
        //uint16_t selector;
        asm volatile("movl %%eax, %0":"=m"(eax));
        asm volatile("movl %%ecx, %0":"=m"(ecx));
        asm volatile("movl %%edx, %0":"=m"(edx));
        asm volatile("movl %%ebx, %0":"=m"(ebx));
        asm volatile("movl %%esi, %0":"=m"(esi));
        asm volatile("movl %%edi, %0":"=m"(edi));
        asm volatile("movl %0, %%eax"::"m"(num));
        asm volatile("movl %0, %%ecx"::"m"(a1));
        asm volatile("movl %0, %%edx"::"m"(a2));
        asm volatile("movl %0, %%ebx"::"m"(a3));
        asm volatile("movl %0, %%esi"::"m"(a4));
        asm volatile("movl %0, %%edi"::"m"(a5));
        asm volatile("int $0x80");
        asm volatile("movl %%eax, %0":"=m"(ret));
        asm volatile("movl %0, %%eax"::"m"(eax));
        asm volatile("movl %0, %%ecx"::"m"(ecx));
        asm volatile("movl %0, %%edx"::"m"(edx));
        asm volatile("movl %0, %%ebx"::"m"(ebx));
        asm volatile("movl %0, %%esi"::"m"(esi));
        asm volatile("movl %0, %%edi"::"m"(edi));

        return ret;
}
```

之后查询IDT表，跳转irqSyscall函数,将中断号压栈后跳转asmDoIrq函数，将其他寄存器压栈（组成StackFrame)，跳转irqHandle函数

```
irqSyscall:
        pushl $0 // push dummy error code
        pushl $0x80 // push interruption number into kernel stack
        jmp asmDoIrq

.global asmDoIrq
asmDoIrq:
        pushal // push process state into kernel stack
        pushl %ds
        pushl %es
        pushl %fs
        pushl %gs
        pushl %esp //esp is treated as a parameter
        call irqHandle
        addl $4, %esp //esp is on top of kernel stack
        popl %gs
        popl %fs
        popl %es
        popl %ds
        popal
        addl $4, %esp //interrupt number is on top of kernel stack
        addl $4, %esp //error code is on top of kernel stack
        iret
```

irqHandle中切换内核态，跳转syscallHandle函数

```c
void irqHandle(struct StackFrame *sf) { // pointer sf = esp
	/* Reassign segment register */
	asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));
	/* Save esp to stackTop */
	uint32_t tmpStackTop = pcb[current].stackTop;
	pcb[current].prevStackTop = pcb[current].stackTop;
	pcb[current].stackTop = (uint32_t)sf;

	switch(sf->irq) {
		case -1:
			break;
		case 0xd:
			GProtectFaultHandle(sf);
			break;
		case 0x20:
			timerHandle(sf);
			break;
		case 0x21:
			keyboardHandle(sf);
			break;
		case 0x80:
			syscallHandle(sf);
			break;
		default:assert(0);
	}
	/* Recover stackTop */
	pcb[current].stackTop = tmpStackTop;
}
```

syscallHandle函数中，根据eax跳转syscallSem函数

```c
void syscallHandle(struct StackFrame *sf) {
        switch(sf->eax) { // syscall number
                case SYS_WRITE:
                        syscallWrite(sf);
                        break; // for SYS_WRITE
                case SYS_READ:
                        syscallRead(sf);
                        break; // for SYS_READ
                case SYS_FORK:
                        syscallFork(sf);
                        break; // for SYS_FORK
                case SYS_EXEC:
                        syscallExec(sf);
                        break; // for SYS_EXEC
                case SYS_SLEEP:
                        syscallSleep(sf);
                        break; // for SYS_SLEEP
                case SYS_EXIT:
                        syscallExit(sf);
                        break; // for SYS_EXIT
                case SYS_SEM:
                        syscallSem(sf);
                        break; // for SYS_SEM
                default:break;
        }
}
```

syscallSem函数中，根据ecx跳转SEM_INIT函数

```
void syscallSem(struct StackFrame *sf) {
	switch(sf->ecx) {
		case SEM_INIT:
			syscallSemInit(sf);
			break;
		case SEM_WAIT:
			syscallSemWait(sf);
			break;
		case SEM_POST:
			syscallSemPost(sf);
			break;
		case SEM_DESTROY:
			syscallSemDestroy(sf);
			break;
		default:break;
	}
}
```

最后在syscallSemInit函数中进行处理

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