

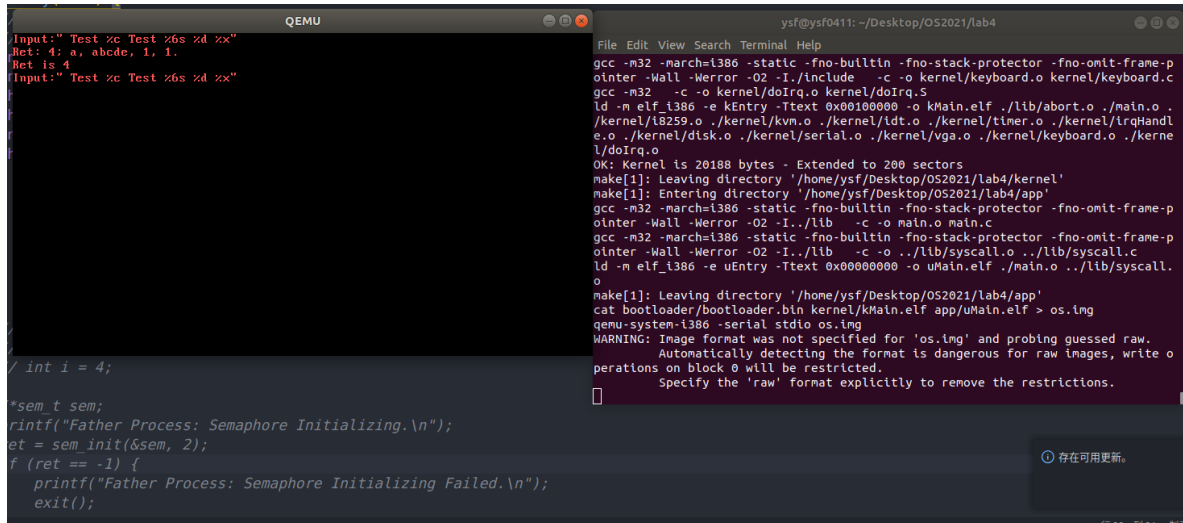
Lab4 实验报告

191300073 杨斯凡 191300073@smail.nju.edu.cn

一、实验进度

我完成了所有必做内容，选做内容完成了生产者消费者问题。

二、实验结果



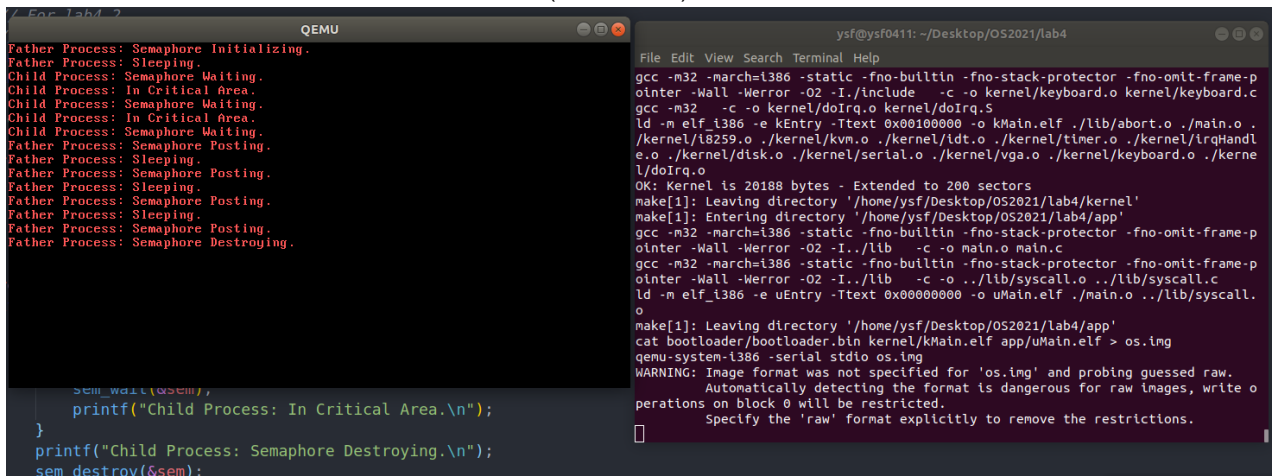
The screenshot shows two windows. The left window is a QEMU terminal with the following output:

```
Input: "Test %c Test %6s %d %x"
Ret: 4; a, abcde, 1, 1.
Ret is 4
Input: "Test %c Test %6s %d %x"
```

The right window is a terminal showing the compilation and linking of the kernel and user space programs. The output includes:

```
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-p
ointer -Wall -Werror -O2 -I./include -c -o kernel/keyboard.o kernel/keyboard.c
gcc -m32 -c -o kernel/doIrq.o kernel/doIrq.S
ld -m elf_i386 -e kEntry -Ttext 0x00100000 -o kMain.elf ./lib/abort.o ./main.o .
./kernel/l8259.o ./kernel/kvm.o ./kernel/ldt.o ./kernel/timer.o ./kernel/irqHandl
e.o ./kernel/disk.o ./kernel/serial.o ./kernel/vga.o ./kernel/keyboard.o ./kerne
l/doIrq.o
OK: Kernel is 20188 bytes - Extended to 200 sectors
make[1]: Leaving directory '/home/ysf/Desktop/OS2021/lab4/kernel'
make[1]: Entering directory '/home/ysf/Desktop/OS2021/lab4/app'
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-p
ointer -Wall -Werror -O2 -I./lib -c -o main.o main.c
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-p
ointer -Wall -Werror -O2 -I./lib -c -o ./lib/syscall.o ./lib/syscall.c
ld -m elf_i386 -e uEntry -Ttext 0x00000000 -o uMain.elf ./main.o ./lib/syscall.
o
make[1]: Leaving directory '/home/ysf/Desktop/OS2021/lab4/app'
cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
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.
```

(scanf 测试)

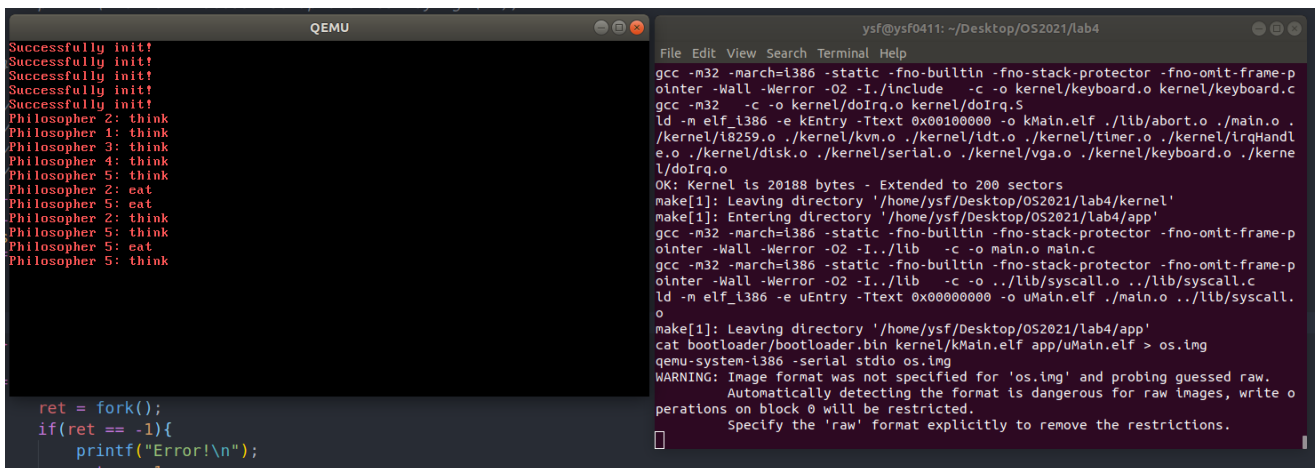


The screenshot shows two windows. The left window is a QEMU terminal with the following output:

```
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.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Destroying.
```

The right window is a terminal showing the compilation and linking of the kernel and user space programs, identical to the previous screenshot.

(信号量测试)



The screenshot shows two windows. The left window is a QEMU terminal with the following output:

```
Successfully init!!
Successfully init!!
Successfully init!!
Successfully init!!
Successfully init!!
Philosopher 2: think
Philosopher 1: think
Philosopher 3: think
Philosopher 4: think
Philosopher 5: think
Philosopher 2: eat
Philosopher 5: eat
Philosopher 2: think
Philosopher 5: think
Philosopher 5: eat
Philosopher 5: think
```

The right window is a terminal showing the compilation and linking of the kernel and user space programs, identical to the previous screenshots.

(哲学家问题)

索引保存在 eax 中。下一次使用信号量即可根据索引来找到该信号量。

然后是信号量的 P 操作。该操作需要申请一个资源。如果此时该资源的值小于 0，则需要等待，于是阻塞该进程。即把该进程挂起在信号量的链表上，然后调用 int\$0x20 指令切换进程。为了和 sleep 区分，把此时 sleeptime 设置为-1。

四、修改代码位置

首先要完成 scanf 函数，也就是 syscallReadStdIn 函数，它完成的工作是。如果 dev[STD_IN].value == 0，将当前进程阻塞在 dev[STD_IN]上。进程被唤醒，读取 keyBuffer 中的字符。

因此我首先模仿教程中的阻塞代码，将当前进程阻塞，然后使用了"int \$0x20"进行进程切换，然后去读取字符缓冲区上的字符，把这些字符传给用户进程，并且把当前进程的 eax 寄存器的值设置为字符数量，作为返回值。

```
asm volatile("int $0x20");//进行调度

int sel = sf->ds;
char *str = (char *)sf->edx;
int size = sf->ebx;//还有个size参数
char character;
int i = 0;
asm volatile("movw %0, %%es:::m"(sel));
while(i < size) {
    if(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++;
        }
    }
    else//缓冲器满了
        break;
}
// asm volatile("movb $0x00, %%es:(%0):::r"(str+i));//\0
pcb[current].regs.eax = i;
return;
```

然后是实现信号量的函数，主要是 syscallSemInit、syscallSemWait、syscallSemPost 和 syscallSemDestroy 四个函数。

对于 syscallSemInit 函数，只需要在信号量数组中找到一个空闲的位置，对这个位置上的信号量进行初始化，并且把信号量数组的索引传给当前进程的 eax 作为返回值，以便于后序对这个信号量的调用。

```

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

```

对于 syscallSemWait, 只需要对给定的信号量的 value 进行减一, 如果大于等于 0 直接返回即可, 如果小于 0, 需要挂起当前进程, 然后使用“int \$0x20”进行进程的切换, 并且返回 0 表示操作成功。

```

void syscallSemWait(struct StackFrame *sf) {
    // TODO: complete `SemWait` and note that you need to consider some special situations
    int num = sf->edx;
    if(sem[num].state == 1){
        sem[num].value--;
        if(sem[num].value >= 0){
            pcb[current].regs.eax = 0;
            return;
        }
        pcb[current].blocked.next = sem[num].pcb.next;
        pcb[current].blocked.prev = &(sem[num].pcb);
        sem[num].pcb.next = &(pcb[current].blocked);
        (pcb[current].blocked.next)->prev = &(pcb[current].blocked);

        pcb[current].sleepTime = -1;
        pcb[current].state = STATE_BLOCKED;
        asm volatile("int $0x20");
        pcb[current].regs.eax = 0;
    }
    else{
        pcb[current].regs.eax = -1;
    }
    return;
}

```

对于 syscallSemWait, 首先需要对给定的信号量加一, 如果大于零直接返回即可, 如果小于 0, 需要把当前信号量上的挂起的进程从队列中移除, 并且设为就绪态即可, 并且返回 0 表示操作成功。

```

void syscallSemPost(struct StackFrame *sf) {
    int i = (int)sf->edx;
    // ProcessTable *pt = NULL;
    if (i < 0 || i >= MAX_SEM_NUM) {
        pcb[current].regs.eax = -1;
        return;
    }
    // TODO: complete other situations
    if(sem[i].state == 1){
        sem[i].value++;
        if(sem[i].value <= 0){
            ProcessTable *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;
        }
        pcb[current].regs.eax = 0;
    }
    else{
        pcb[current].regs.eax = -1;
    }
    return;
}

```

对于 sysSemDestory, 只需要把当前的信号量的 state 设为 0, value 设为 0, 返回 0 即可。

```

void syscallSemDestroy(struct StackFrame *sf) {
    // TODO: complete `SemDestroy`
    int destroy = sf->edx;
    if(sem[destroy].state == 1){
        sem[destroy].state = 0;
        sem[destroy].value = 0;
        pcb[current].regs.eax = 0;
    }
    else{
        pcb[current].regs.eax = -1;
    }
    return;
}

```

对于哲学家问题, 按照教程上的伪代码完成即可, 只需要每次 fork 出子进程即可。实现 getpid 函数, 只需要仿照其他的系统调用, 处理函数为 syscallGetPid, 只需要将 eax 的值设为 current 即可。

但是, 有一个问题, 就是框架代码仅仅支持 4 个进程和 4 个信号量同时工作, 但是哲学家问题有 5 个进程和信号量, 因此需要对框架代码进行修改, 我首先将信号量的数目设为了 8, 然后将进程的数目设为了 9, 修改进程数目只需要修改 GDT size 即可。

五、bug 及其解决

在实现读者写者问题的时候, 我发现每次都会中途停下, 在我进行了很久的调试之后, 发现 count 的值出现了问题, 然后我分析原因, 发现 count 在每个进程之间独立, 每次进行修改的时候, 只对本进程的 count 进行了修改, 其他进程的 count 还是 0, 这就导致了 bug 的出现。

于是我在内核中定义了一个变量 memory, 作为 count, 然后每次更改的之后将更改的值赋予 memory, 然后读取的时候读取 memory 作为 count 的值进行操作, 这样就可以解决这个问题了。

```

void syscallReadMemory(struct StackFrame *sf){
    pcb[current].regs.eax = memory;
    return;
}

void syscallWriteMemory(struct StackFrame *sf){
    memory= (int)(sf->edx);
    pcb[current].regs.eax = memory;
    return;
}

```

```

int write(int fd,int num) {
    return syscall(SYS_WRITE, fd, num, 0, 0, 0);
}

int read(int fd) {
    return syscall(SYS_READ, fd, 0,0,0,0);
}

```