Lab4 进程同步

201300032 肖思远

实验环境

Ubuntu 21.04 + gcc 10.3.0

实验目的

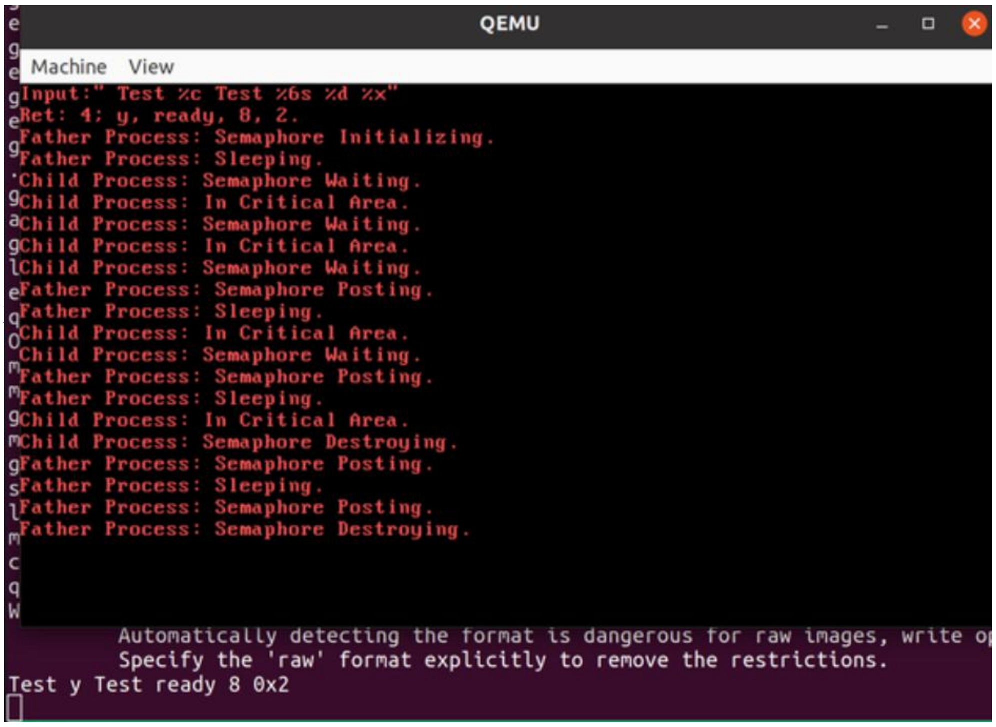
1. 实现格式化输入函数
2. 实现 4 个信号量相关系统调用
3. 实现哲学家就餐问题
4. 实现生产者-消费者问题，读者写者问题（选做）

实验进度

我完成了全部内容，包括两个选做问题

实验结果

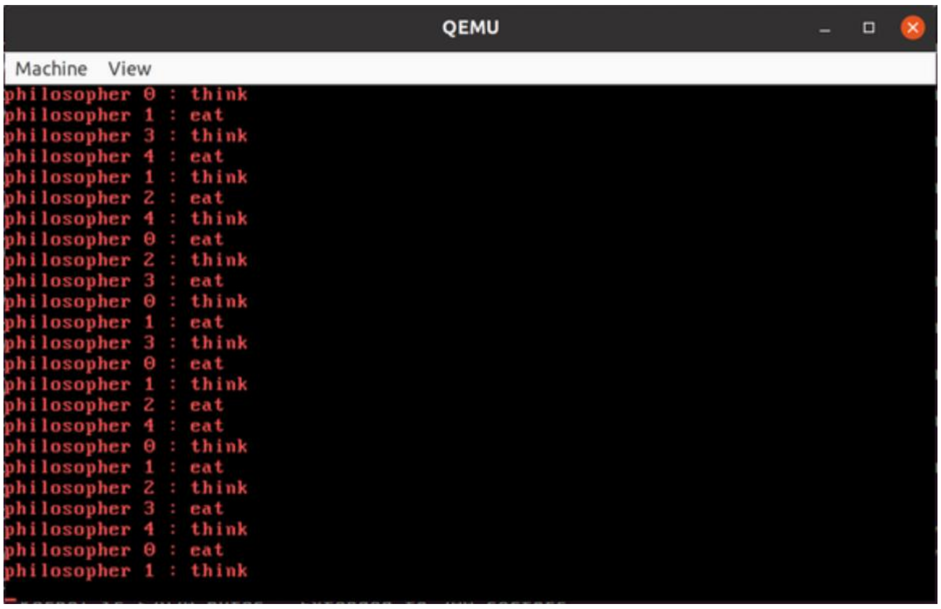
4.1 与 4.2（注：最下面串口显示了我的输入）



```
Input:" Test %c Test %6s %d %x"
Ret: 4; y, ready, 8, 2.
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.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
```

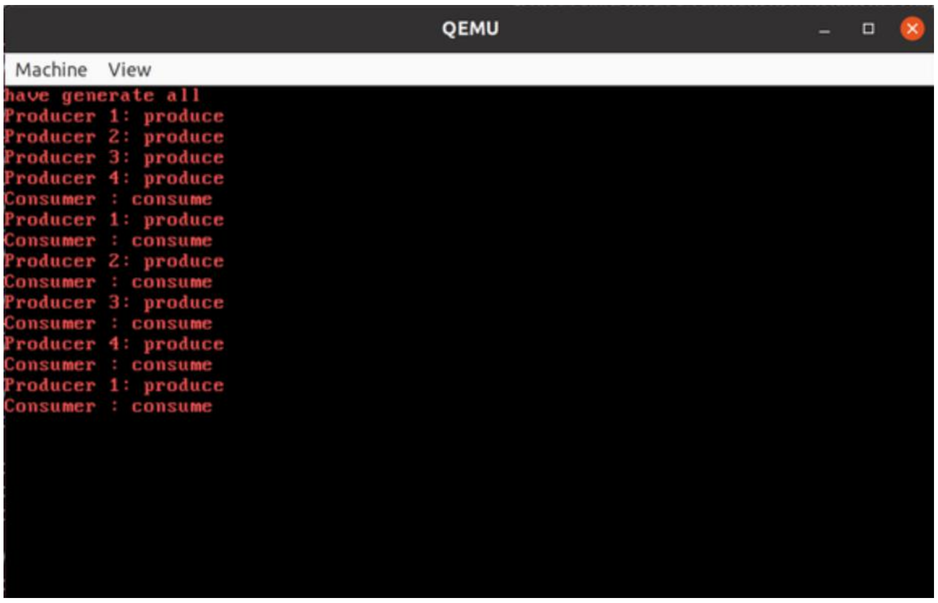
```
        Automatically detecting the format is dangerous for raw images, write op
        Specify the 'raw' format explicitly to remove the restrictions.
Test y Test ready 8 0x2
```

4.3 哲学家就餐问题



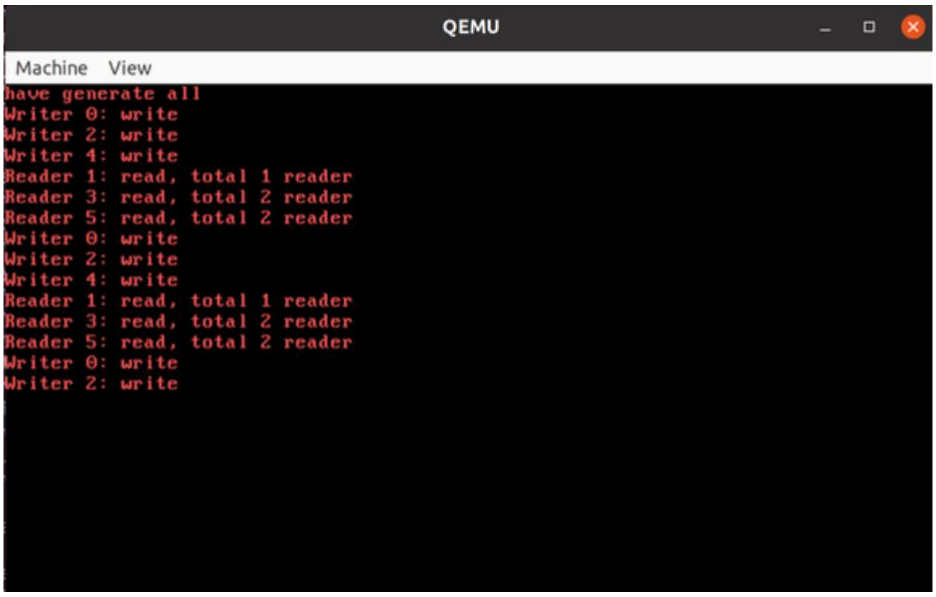
```
philosopher 0 : think
philosopher 1 : eat
philosopher 3 : think
philosopher 4 : eat
philosopher 1 : think
philosopher 2 : eat
philosopher 4 : think
philosopher 0 : eat
philosopher 2 : think
philosopher 3 : eat
philosopher 0 : think
philosopher 1 : eat
philosopher 3 : think
philosopher 0 : eat
philosopher 1 : think
philosopher 2 : eat
philosopher 4 : eat
philosopher 0 : think
philosopher 1 : eat
philosopher 2 : think
philosopher 3 : eat
philosopher 4 : think
philosopher 0 : eat
philosopher 1 : think
```

选做 1：生产者与消费者



```
have generate all
Producer 1: produce
Producer 2: produce
Producer 3: produce
Producer 4: produce
Consumer : consume
Producer 1: produce
Consumer : consume
Producer 2: produce
Consumer : consume
Producer 3: produce
Consumer : consume
Producer 4: produce
Consumer : consume
Producer 1: produce
Consumer : consume
```

选做 2：读者-写者

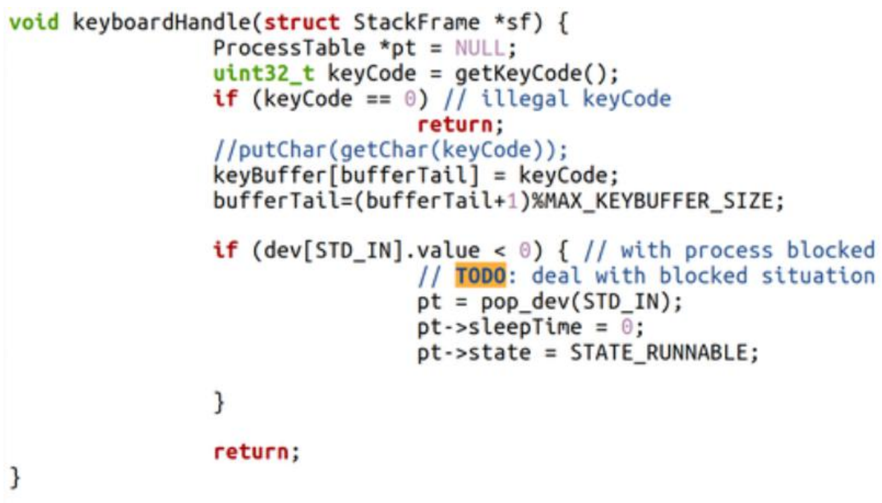


```
have generate all
Writer 0: write
Writer 2: write
Writer 4: write
Reader 1: read, total 1 reader
Reader 3: read, total 2 reader
Reader 5: read, total 2 reader
Writer 0: write
Writer 2: write
Writer 4: write
Reader 1: read, total 1 reader
Reader 3: read, total 2 reader
Reader 5: read, total 2 reader
Writer 0: write
Writer 2: write
```

关键代码实现方法

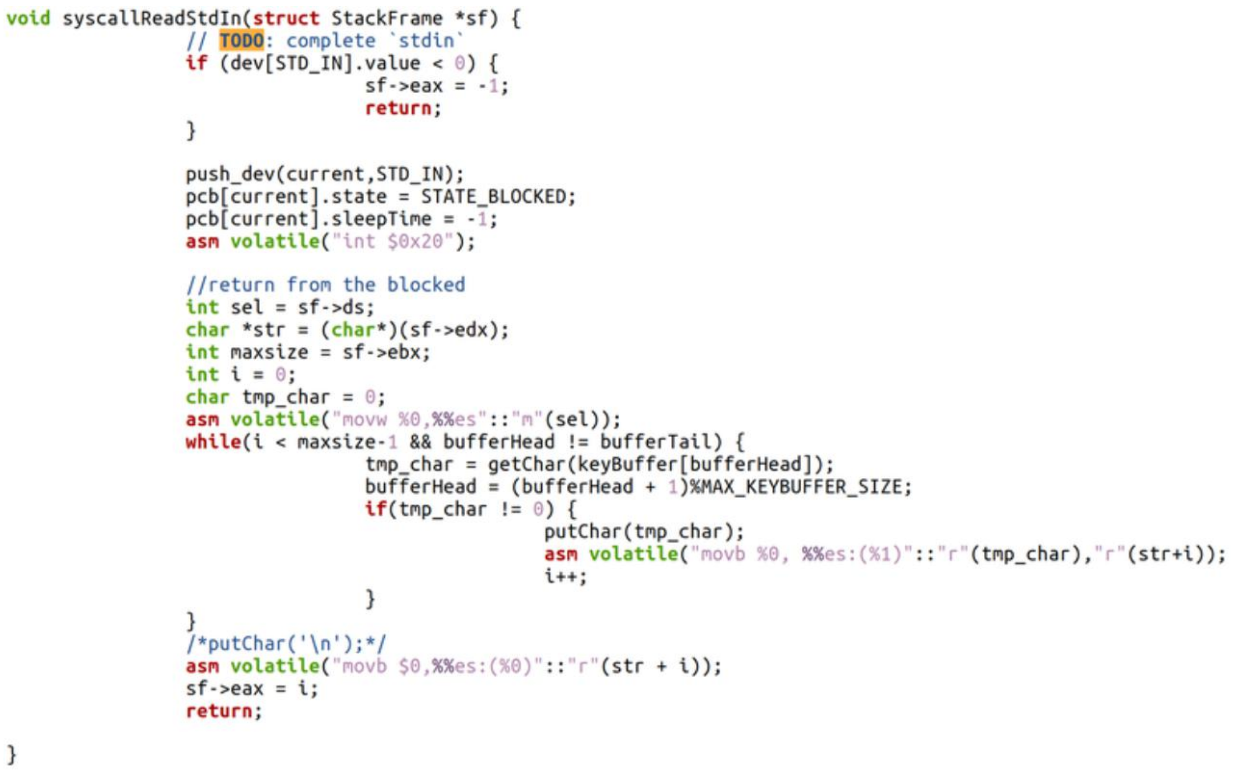
注：实验手册上提供加入线程到阻塞列表与从阻塞列表中取出被我抽象成 4 个函数（pop_dev 函数 ，push_dev 函数 ，pop_sem 函数，push_sem 函数）

- **keyboardHandle( )**



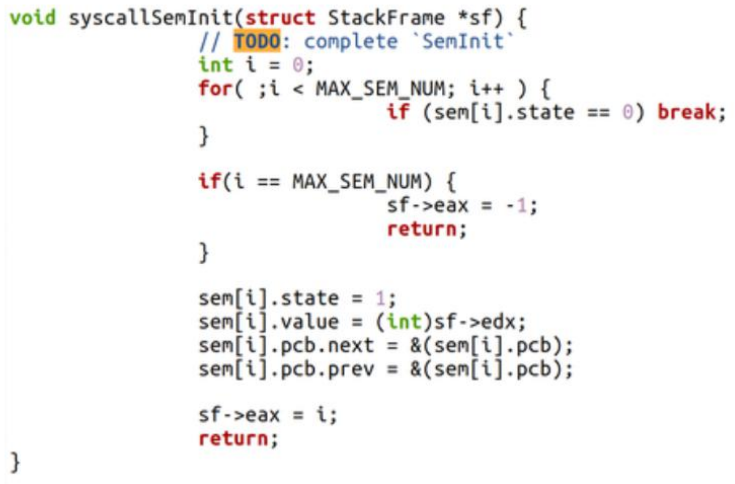
```
void keyboardHandle(struct StackFrame *sf) {
                ProcessTable *pt = NULL;
                uint32_t keyCode = getKeyCode();
                if (keyCode == 0) // illegal keyCode
                                return;
                //putChar(getChar(keyCode));
                keyBuffer[bufferTail] = keyCode;
                bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;

                if (dev[STD_IN].value < 0) { // with process blocked
                                // TODO: deal with blocked situation
                                pt = pop_dev(STD_IN);
                                pt->sleepTime = 0;
                                pt->state = STATE_RUNNABLE;

                }

                return;
}
```

• **syscallReadStdIn( )**



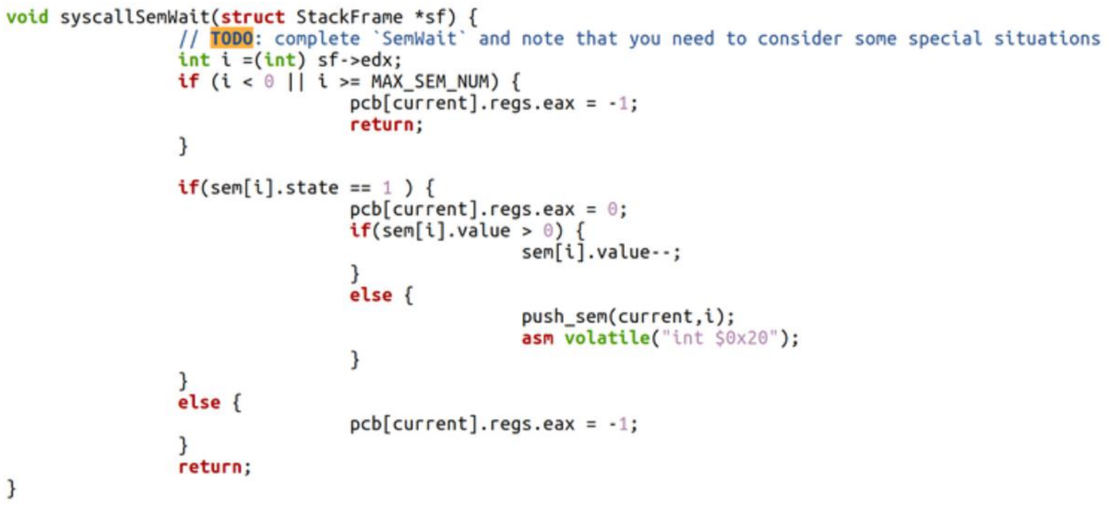
```c
void syscallReadStdIn(struct StackFrame *sf) {
                // TODO: complete `stdin`
                if (dev[STD_IN].value < 0) {
                                sf->eax = -1;
                                return;
                }

                push_dev(current,STD_IN);
                pcb[current].state = STATE_BLOCKED;
                pcb[current].sleepTime = -1;
                asm volatile("int $0x20");

                //return from the blocked
                int sel = sf->ds;
                char *str = (char*)(sf->edx);
                int maxsize = sf->ebx;
                int i = 0;
                char tmp_char = 0;
                asm volatile("movw %0,%%es"::"m"(sel));
                while(i < maxsize-1 && bufferHead != bufferTail) {
                                tmp_char = getChar(keyBuffer[bufferHead]);
                                bufferHead = (bufferHead + 1)%MAX_KEYBUFFER_SIZE;
                                if(tmp_char != 0) {
                                                putChar(tmp_char);
                                                asm volatile("movb %0, %%es:(%1)"::"r"(tmp_char),"r"(str+i));
                                                i++;
                                }
                }
                /*putChar('\n');*/
                asm volatile("movb $0,%%es:(%0)"::"r"(str + i));
                sf->eax = i;
                return;

}
```

• **syscallSemInit( )**



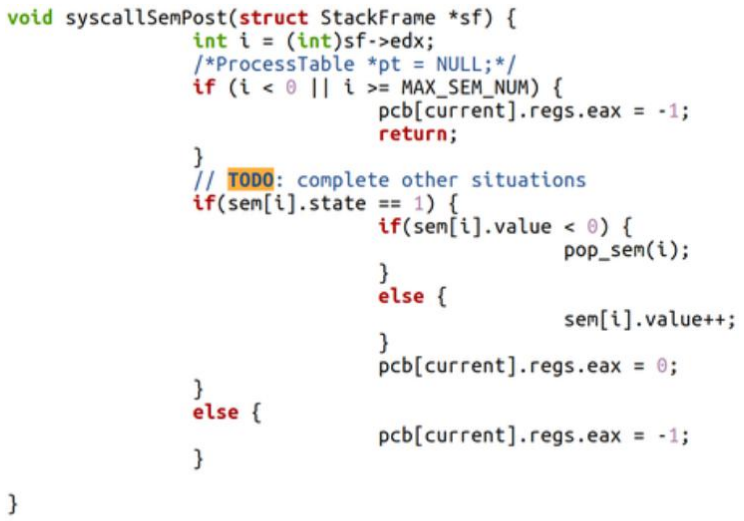
```c
void syscallSemInit(struct StackFrame *sf) {
                // TODO: complete `SemInit`
                int i = 0;
                for( ;i < MAX_SEM_NUM; i++ ) {
                                if (sem[i].state == 0) break;
                }

                if(i == MAX_SEM_NUM) {
                                sf->eax = -1;
                                return;
                }

                sem[i].state = 1;
                sem[i].value = (int)sf->edx;
                sem[i].pcb.next = &(sem[i].pcb);
                sem[i].pcb.prev = &(sem[i].pcb);

                sf->eax = i;
                return;
}
```

- **syscallSemWait( )**



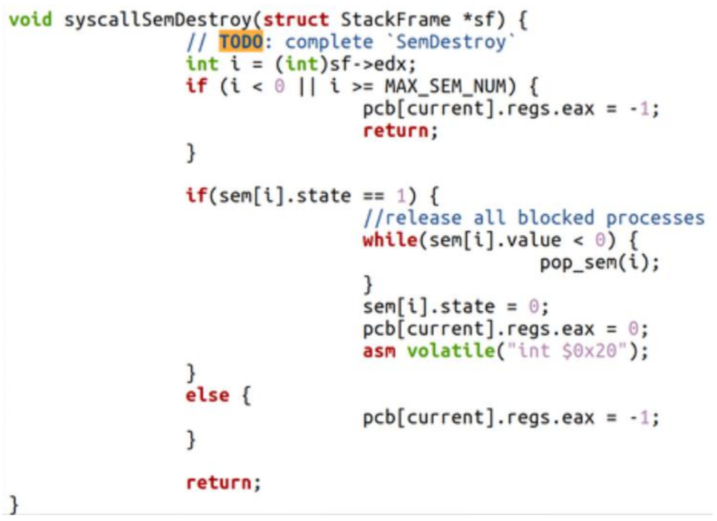
```
void syscallSemWait(struct StackFrame *sf) {
	// TODO: complete `SemWait` and note that you need to consider some special situations
	int i =(int) sf->edx;
	if (i < 0 || i >= MAX_SEM_NUM) {
		pcb[current].regs.eax = -1;
		return;
	}

	if(sem[i].state == 1 ) {
		pcb[current].regs.eax = 0;
		if(sem[i].value > 0) {
			sem[i].value--;
		}
		else {
			push_sem(current,i);
			asm volatile("int $0x20");
		}
	}
	else {
		pcb[current].regs.eax = -1;
	}
	return;
}
```

- **syscallSemPost( )**



```
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) {
		if(sem[i].value < 0) {
			pop_sem(i);
		}
		else {
			sem[i].value++;
		}
		pcb[current].regs.eax = 0;
	}
	else {
		pcb[current].regs.eax = -1;
	}

}
```

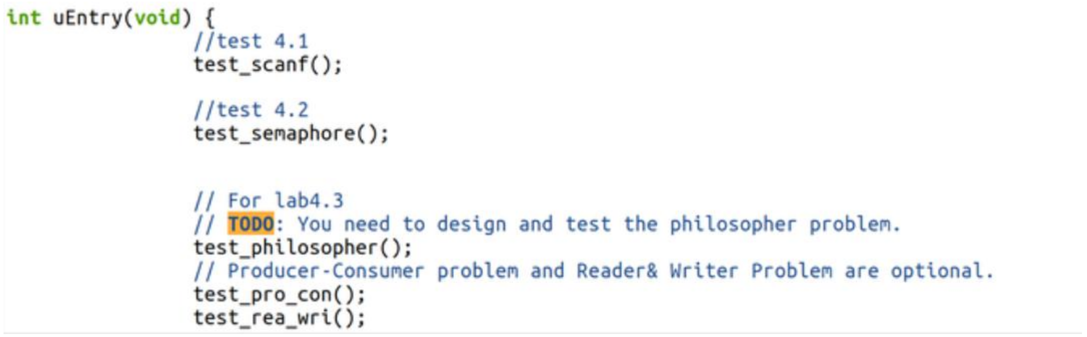
syscallSemDestroy()



```c
void syscallSemDestroy(struct StackFrame *sf) {
		// TODO: complete `SemDestroy`
		int i = (int)sf->edx;
		if (i < 0 || i >= MAX_SEM_NUM) {
				pcb[current].regs.eax = -1;
				return;
		}

		if(sem[i].state == 1) {
				//release all blocked processes
				while(sem[i].value < 0) {
						pop_sem(i);
				}
				sem[i].state = 0;
				pcb[current].regs.eax = 0;
				asm volatile("int $0x20");
		}
		else {
				pcb[current].regs.eax = -1;
		}

		return;
}
```

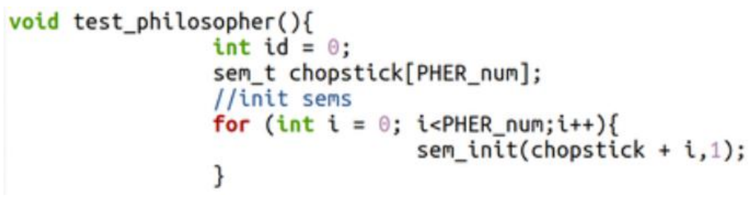
用户程序的封装

在用户程序中，我将不同的任务的测试进行了封装，如下图所示。各个任务产生的进程行为有差异，使进程消亡较为麻烦，故每次测试只测一个任务，其余任务被注释掉。

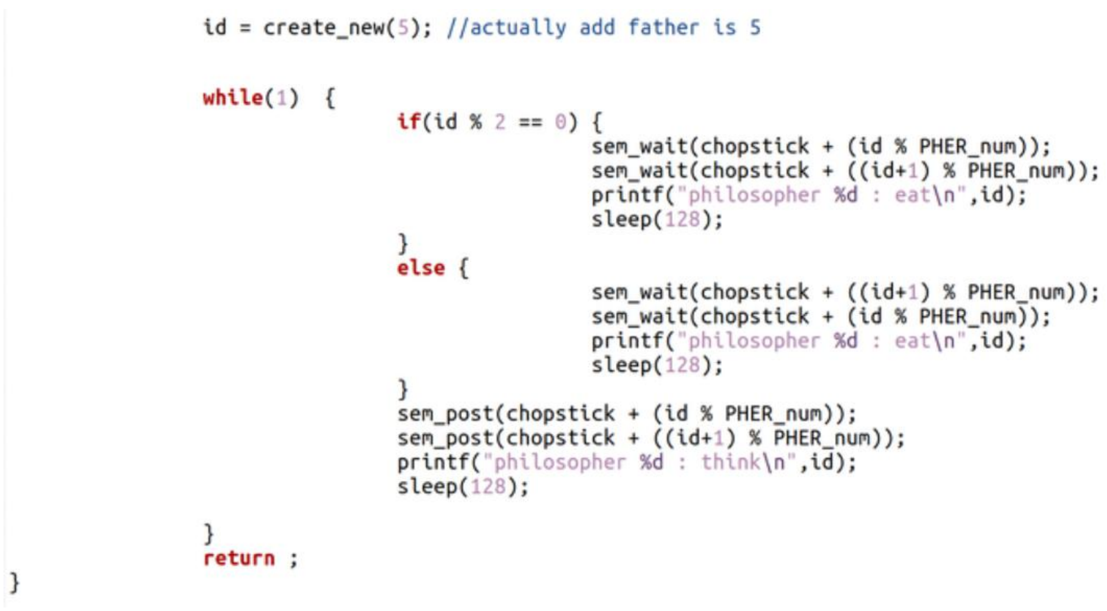


```c
int uEntry(void) {
		//test 4.1
		test_scanf();

		//test 4.2
		test_semaphore();


		// For lab4.3
		// TODO: You need to design and test the philosopher problem.
		test_philosopher();
		// Producer-Consumer problem and Reader& Writer Problem are optional.
		test_pro_con();
		test_rea_wri();
```

哲学家就餐问题

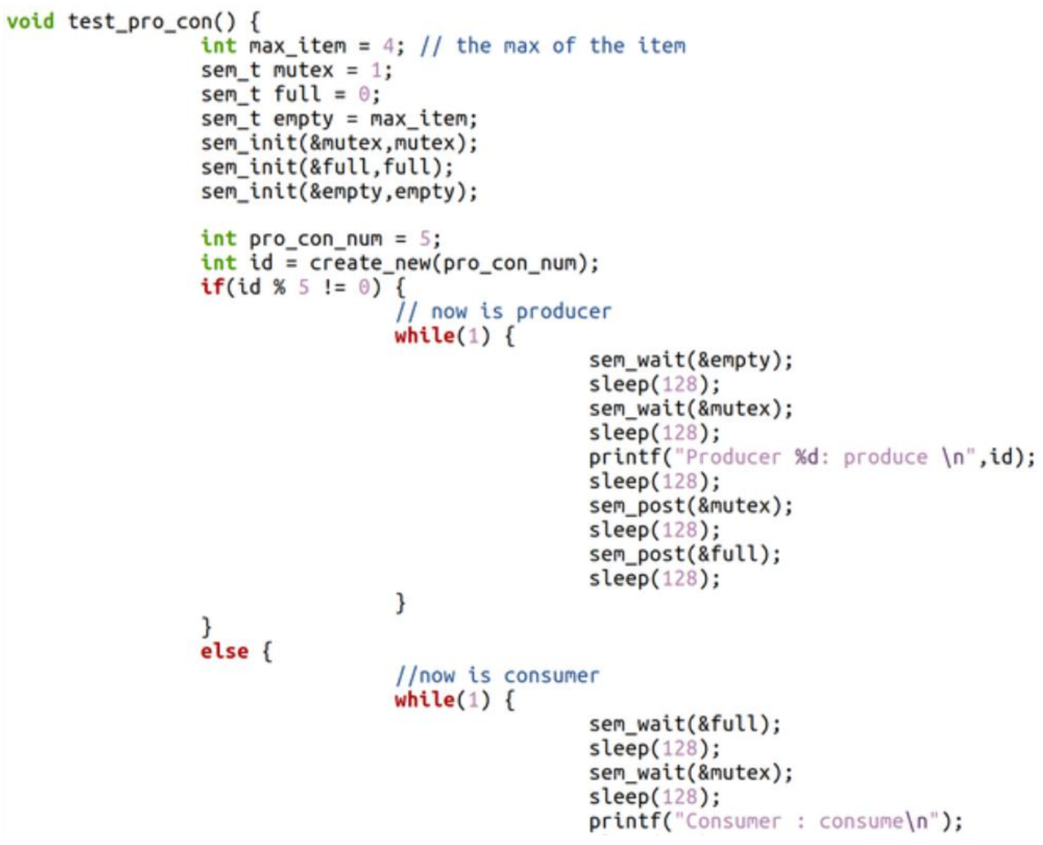


```c
void test_philosopher(){
		int id = 0;
		sem_t chopstick[PHER_num];
		//init sems
		for (int i = 0; i<PHER_num;i++){
				sem_init(chopstick + i,1);
		}
```

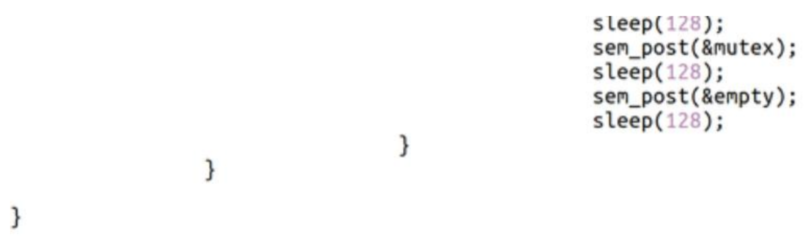


```
        id = create_new(5); //actually add father is 5

        while(1)  {
                if(id % 2 == 0) {
                        sem_wait(chopstick + (id % PHER_num));
                        sem_wait(chopstick + ((id+1) % PHER_num));
                        printf("philosopher %d : eat\n",id);
                        sleep(128);
                }
                else {
                        sem_wait(chopstick + ((id+1) % PHER_num));
                        sem_wait(chopstick + (id % PHER_num));
                        printf("philosopher %d : eat\n",id);
                        sleep(128);
                }
                sem_post(chopstick + (id % PHER_num));
                sem_post(chopstick + ((id+1) % PHER_num));
                printf("philosopher %d : think\n",id);
                sleep(128);

        }
        return ;
}
```

生产者-消费者问题

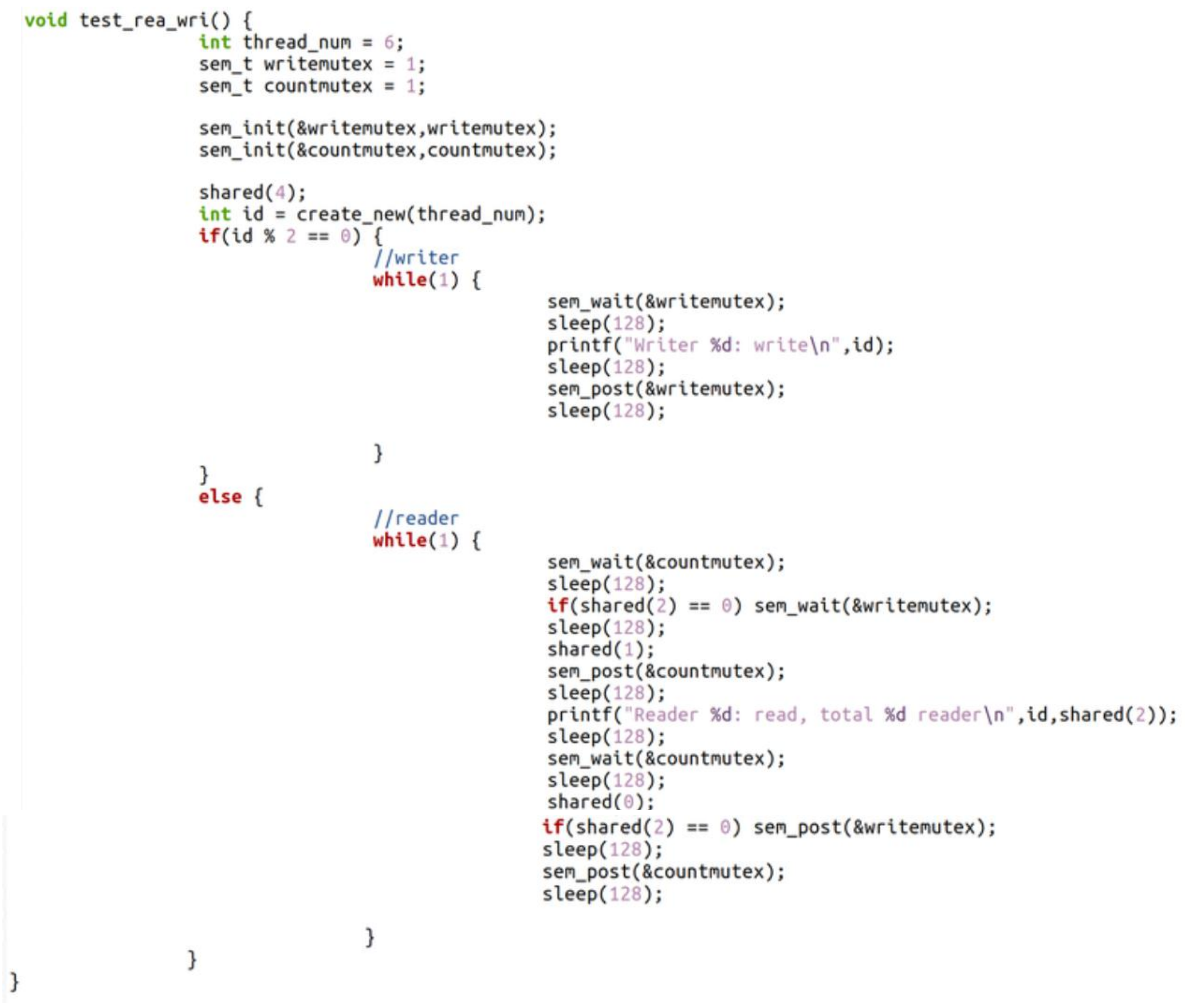


```
void test_pro_con() {
        int max_item = 4; // the max of the item
        sem_t mutex = 1;
        sem_t full = 0;
        sem_t empty = max_item;
        sem_init(&mutex,mutex);
        sem_init(&full,full);
        sem_init(&empty,empty);

        int pro_con_num = 5;
        int id = create_new(pro_con_num);
        if(id % 5 != 0) {
                // now is producer
                while(1) {
                        sem_wait(&empty);
                        sleep(128);
                        sem_wait(&mutex);
                        sleep(128);
                        printf("Producer %d: produce \n",id);
                        sleep(128);
                        sem_post(&mutex);
                        sleep(128);
                        sem_post(&full);
                        sleep(128);
                }
        }
        else {
                //now is consumer
                while(1) {
                        sem_wait(&full);
                        sleep(128);
                        sem_wait(&mutex);
                        sleep(128);
                        printf("Consumer : consume\n");
```



```
            sleep(128);
            sem_post(&mutex);
            sleep(128);
            sem_post(&empty);
            sleep(128);
        }
    }
}
```

读者-写者问题

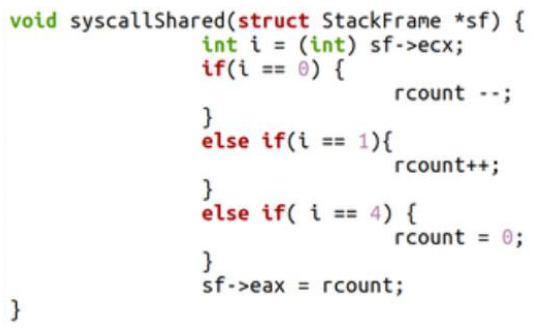


```
void test_rea_wri() {
    int thread_num = 6;
    sem_t writemutex = 1;
    sem_t countmutex = 1;

    sem_init(&writemutex,writemutex);
    sem_init(&countmutex,countmutex);

    shared(4);
    int id = create_new(thread_num);
    if(id % 2 == 0) {
        //writer
        while(1) {
            sem_wait(&writemutex);
            sleep(128);
            printf("Writer %d: write\n",id);
            sleep(128);
            sem_post(&writemutex);
            sleep(128);

        }
    }
    else {
        //reader
        while(1) {
            sem_wait(&countmutex);
            sleep(128);
            if(shared(2) == 0) sem_wait(&writemutex);
            sleep(128);
            shared(1);
            sem_post(&countmutex);
            sleep(128);
            printf("Reader %d: read, total %d reader\n",id,shared(2));
            sleep(128);
            sem_wait(&countmutex);
            sleep(128);
            shared(0);
            if(shared(2) == 0) sem_post(&writemutex);
            sleep(128);
            sem_post(&countmutex);
            sleep(128);

        }
    }
}
```

读者-写者问题中新增加的库函数 shared

由于不同进程有不同的地址空间，为了能让所有进程访问同一块地址空间，我实现了一个库函数 shared。shared 中会调用我实现的新系统调用：syscallshared，进入内核，访问内核中的一块空间（其实仅访问内核中的 int 变量 rcount）。shared 函数行为较简单：shared(0)表示 rcount - -，shared(1)表示 rcount++，shared(2)返回值为 rcount 且不对 rcount 造成任何改变，shared(4)使 rcount 为 0。

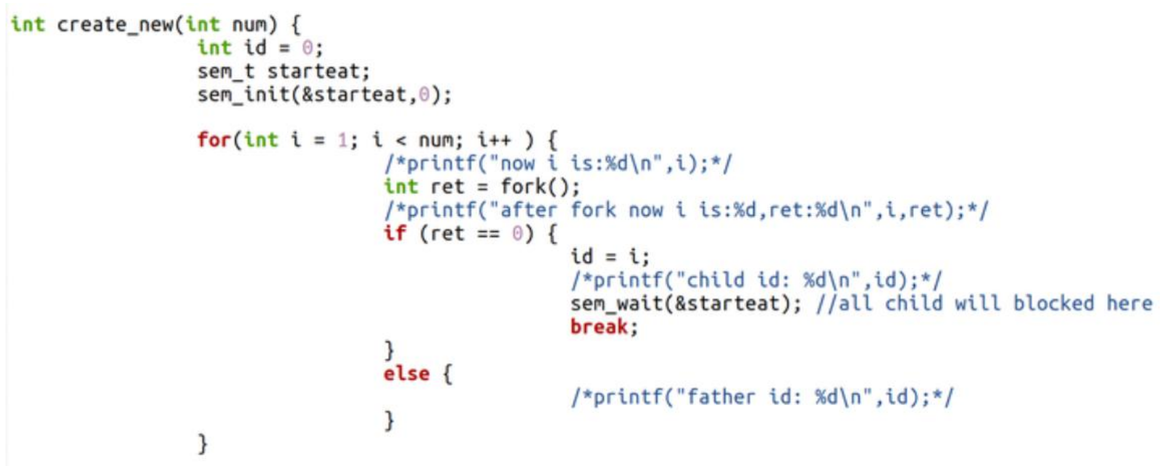
内核中 syscallShared 的代码



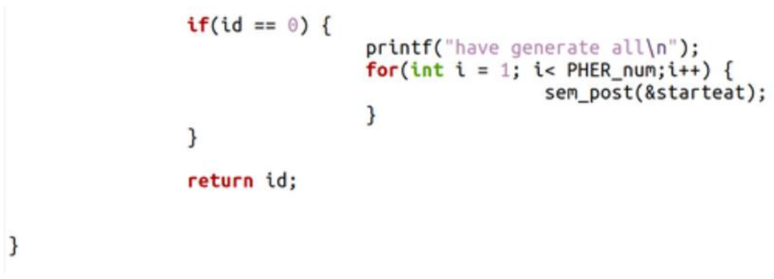
```
void syscallShared(struct StackFrame *sf) {
                int i = (int) sf->ecx;
                if(i == 0) {
                                rcount --;
                }
                else if(i == 1){
                                rcount++;
                }
                else if( i == 4) {
                                rcount = 0;
                }
                sf->eax = rcount;
}
```

产生新进程的封装函数 create_new

在哲学家吃饭问题、生产者消费者问题、读者-写者问题中，产生新进程都使用了一个 create_new 函数。此函数的定义为 int create_new(int num)，其中 num 是希望调用此函数后一共有的用户进程数目（比如生产者消费者是 6 个），返回的是可以区分进程的 id（不是 pid，而是从 0 到 num-1 的 id，这样如果以后多个测试同时进行（比如哲学家吃饭和生产者消费者都进行），各个测试中 id 计数也是分开的，更好看些）



```
int create_new(int num) {
                int id = 0;
                sem_t starteat;
                sem_init(&starteat,0);

                for(int i = 1; i < num; i++ ) {
                                /*printf("now i is:%d\n",i);*/
                                int ret = fork();
                                /*printf("after fork now i is:%d,ret:%d\n",i,ret);*/
                                if (ret == 0) {
                                                id = i;
                                                /*printf("child id: %d\n",id);*/
                                                sem_wait(&starteat); //all child will blocked here
                                                break;
                                }
                                else {
                                                /*printf("father id: %d\n",id);*/
                                }
                }
```



```
                if(id == 0) {
                        printf("have generate all\n");
                        for(int i = 1; i< PHER_num;i++) {
                                sem_post(&starteat);
                        }
                }
                return id;

}
```

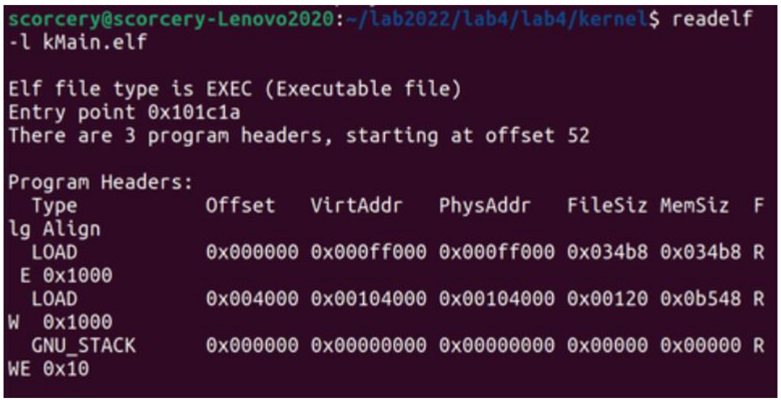
实验思考题

本次实验中没有思考题。

实验中遇到的问题

1. Bootloader 太大，故 CFLAGS 中加了-fno-asynchronous-unwind-tables

2. bootloader 无法加载 kernel，注释掉根据程序头表获得 offset 等内容后，才可加载。用 readelf -l 查看 offset 内容，发现是：0x0，很明显不是正确的.text 开始位置。



```
scorcery@scorcery-Lenovo2020:~/lab2022/lab4/lab4/kernel$ readelf
-l kMain.elf

Elf file type is EXEC (Executable file)
Entry point 0x101c1a
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  F
lg Align
  LOAD           0x000000 0x000ff000 0x000ff000 0x034b8 0x034b8 R
 E 0x1000
  LOAD           0x004000 0x00104000 0x00104000 0x00120 0x0b548 R
W  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 R
WE 0x10
```

我对实验的思考

1. 在实现生产者与消费者时，由于不同进程之中的缓冲区实现起来不太方便，故我用信号量来充当不同进程直接的共享。而由于信号量是陷入内核中获得的，故其互斥性是有所保证的，我认为可以不用 mutex。不过在我的代码中，还是按照伪代码完成，用了 mutex。

2. 实验手册上信号量的阻塞列表的插入与取出写的好漂亮（个人感受）。刚开始看的时候有些看不懂，然后一行一行代码捋发现真的好漂亮，利用了 blocked 在 pcb 中的位置，如果知道了某个 blocked，就可以获得相应的 pcb，所以在那个阻塞列表之中，只用存有关 blocked 的列表，索引时在找到 pcb，就有一种很抽象的感觉（可能是因为，在某种层面上说链表中没有参杂结点自己数据）