课 程 实 验 报 告

实验名称　进程同步与通信

课程名称	操作系统
院　　系	计算机科学与技术系
学　　号	191220129
姓　　名	邢尚禹
邮　　箱	191220129@smail.nju.edu.cn
实验日期	2021 年 5 月

目录

1 实验进度和批改注意事项

已完成所有内容，包括选做。

批改时希望能关注这一点：框架代码的 bootloader **无法在我的环境中正确运行**（ubuntu20.04 + gcc 9.3），必须注释 bootMain 函数的以下两行（第 16，17 行）：

```
1  void bootMain(void)
2  {
3          int i = 0;
4          // int phoff = 0x34;
5          int offset = 0x1000;
6          unsigned int elf = 0x100000;
7          void (*kMainEntry)(void);
8          kMainEntry = (void (*)(void))0x100000;
9
10         for (i = 0; i < 200; i++)
11         {
12                 readSect((void *)(elf + i * 512), 1 + i);
13         }
14
15         kMainEntry = (void (*)(void))((struct ELFHeader *)elf)->entry;
16         // phoff = ((struct ELFHeader *)elf)->phoff;
17         // offset = ((struct ProgramHeader *)(elf + phoff))->off;
18
19         for (i = 0; i < 200 * 512; i++)
20         {
21                 *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
22         }
23
24         kMainEntry();
25 }
```

我猜想可能是由于 gcc 版本的问题，如果批改时不能正确运行，请将上述两行取消注释再尝试编译。

另外，我在 app/main.c 中封装了几个测试函数，可以通过注释和取消注释实现对不同内容的测试（包括选做的内容）。

```c
int uEntry(void)
{
        // For lab4.1
        test_scanf();

        // For lab4.2
        test_sem();

        // For lab4.3
        //! note that this function is non-stopping
        test_philosopher();

        // Optional
        //! note that these functions are non-stopping
        test_producer_consumer();
        test_reader_writer();

        return 0;
}
```

注意，lab4.3 节和选做实现的函数是不会主动退出的，要先注释再重新编译运行才能测试之后的函数。

2 实验思路和过程

2.1 scanf

在 keyboardHandle 中，如果有进程因为 dev[std_in] 阻塞，要释放该进程。

```c
void keyboardHandle(struct StackFrame *sf)
{
        uint32_t keyCode = getKeyCode();
        if (keyCode == 0) // illegal keyCode
                return;
```

```c

        keyBuffer[bufferTail] = keyCode;
        bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;

        assert(dev[STD_IN].value == -1 || dev[STD_IN].value == 0);
        if (dev[STD_IN].value < 0)
        { // with process blocked
                // TODO: deal with blocked situation
                ProcessTable *pt = pop_dev(STD_IN);
                ++dev[STD_IN].value;
                pt->state = STATE_RUNNABLE;
        }

        return;
}
```

此处,我将框架代码给出的阻塞和释放进程的实现封装成了函数 push_sem 和 pop_sem 方便后续使用，并类似实现了函数 push_dev 和 pop_dev。

在 syscallReadStdIn 中，要完成以下的处理：

1. 如果已经有进程在读，直接返回-1；
2. 将进程阻塞在 stdin 上；
3. 被唤醒后，将输入缓冲区拷贝到传入的 buffer。

由于这里是内核代码，用中断嵌套来实现阻塞非常复杂。可以直接强制切换进程，即模拟一个时钟中断信号，转到 timerHandle 中处理。被唤醒后，将自动接着运行。

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

```
        push_dev(current, STD_IN);
        pcb[current].state = STATE_BLOCKED;
        asm volatile("int $0x20");
        int sel = sf->ds;
        char *str = (char *)sf->edx;
        int size = sf->ebx;
        int i = 0;
        char character;
        asm volatile("movw %0, %%es" ::"m"(sel));
        while (i < size - 1 && bufferHead != bufferTail)
        {
                character = getChar(keyBuffer[bufferHead]);
                bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
                if (character != 0)
                {
                        stdout_char(character);
                        asm volatile("movb %0, %%es:(%1)" ::"r"(character), "r"(str + i));
                        ++i;
                }
        }
        asm volatile("movb $0, %%es:(%0)" ::"r"(str + i));
        sf->eax = i;
}
```

2.2 信号量

init 和 destroy 非常简单，此处不再赘述。对 wait 函数，要先将信号量-1，如果 <0 则将当前进程阻塞在 sem 上，然后模拟时钟中断切换进程。

```
void syscallSemWait(struct StackFrame *sf)
{
        int i = (int)sf->edx;
        if (i < 0 || i >= MAX_SEM_NUM)
        {
                pcb[current].regs.eax = -1;
                return;
```

```c
        }
        if (--sem[i].value < 0)
        {
                push_sem(current, i);
                // ++pcb[current].blocked_sems;
                pcb[current].state = STATE_BLOCKED;
                asm volatile("int $0x20");
        }
        sf->eax = 0;
}
```

对 post 函数，先将信号量 +1，如果 <=0 则从该信号阻塞的表中取出一项，设置为 RUNNABLE。

```c
void syscallSemPost(struct StackFrame *sf)
{
    int i = (int)sf->edx;
    if (i < 0 || i >= MAX_SEM_NUM)
    {
        pcb[current].regs.eax = -1;
        return;
    }
    if (++sem[i].value <= 0)
    {
        ProcessTable *pt = pop_sem(i);
        // if (!--pt->blocked_sems)
        pt->state = STATE_RUNNABLE;
    }
    sf->eax = 0;
}
```

2.3 进程同步与通信

这一节的主要目的是测试信号量是否实现正确。为了更好地完成测试，我认为每次 sleep(128) 并不好，因为每次固定间隔可能导致有些内容测试不到。因此我自己写了 rand 函数，每次 sleep(rand() % 128)，这样测试更加全面。固定 seed 可以使结果可复现，不会对 debug 造成影响。

```c
uint32_t next = 0;
uint32_t rand()
{
        return next = next * 1103515245 + 12345;
}
```

这里实现的 rand 和 C 标准库的不一样，主要区别是没有加锁，多个进程同时调用可能多次返回一个未更新的结果。这样可能导致如果调用次数较少，rand 返回值不再是随机均匀采样。但这个问题在本次实验中不重要，因为我们没有随机均匀采样的要求。

为了完成测试，还需要实现 get_pid 系统调用。实现非常简单，补全系统调用，在中断处理程序中直接返回 current 即可。代码此处省略。

另外，这里的进程数会 >4，因此要修改 memery.h 中的 NR_SEGMENT 值，这里将其设置为 20，即允许 9 个进程。

2.3.1 哲学家问题

通过 fork 函数产生子进程，一共 5 个进程分别调用 philosopher 过程即可。

```c
#define N 5 // number of philosopher
void philosopher(int i, sem_t *forks)
{
        --i; // pid start at 1
        while (1)
        {
                if (i % 2 == 0)
                {
                        sem_wait(&forks[i]);
                        sleep(rand() % 128);
                        sem_wait(&forks[(i + 1) % N]);
                }
                else
                {
                        sem_wait(&forks[(i + 1) % N]);
                        sleep(rand() % 128);
                        sem_wait(&forks[i]);
```

```
		}
		printf("Philosopher %d: eat\n", i);
		sleep(rand() % 128 + 64); // eat
		printf("Philosopher %d: think\n", i);
		sem_post(&forks[i]);
		sleep(rand() % 128);
		sem_post(&forks[(i + 1) % N]);
		sleep(rand() % 128 + 64); // think
	}
}

void test_philosopher()
{
	sem_t forks[N];
	int id;
	for (int i = 0; i < N; ++i)
		sem_init(&forks[i], 1);
	for (int i = 0; i < N - 1; ++i)
	{
		int ret = fork();
		if (ret == -1)
		{
			printf("fork error: %d\n", i);
			exit();
		}
		if (ret == 0) // child
		{
			id = get_pid();
			philosopher(id, forks);
		}
	}
	id = get_pid();
	philosopher(id, forks);
}
```

2.3.2 生产者消费者问题

通过 fork 函数产生 4 个子进程，让所有的子进程调用 producer 过程，父进程调用 consumer 过程。

```c
void producer(int id, sem_t *mutex, sem_t *full, sem_t *empty)
{
        --id; // pid start at 1
        while (1)
        {
                sem_wait(empty);
                sleep(rand() % 128);
                sem_wait(mutex);
                sleep(rand() % 128);
                printf("Producer %d: produce\n", id);
                sleep(rand() % 128);
                sem_post(mutex);
                sleep(rand() % 128);
                sem_post(full);
                sleep(rand() % 128);
        }
}

void consumer(sem_t *mutex, sem_t *full, sem_t *empty)
{
        while (1)
        {
                sem_wait(full);
                sleep(rand() % 128);
                sem_wait(mutex);
                sleep(rand() % 128);
                printf("Consumer: consume\n");
                sleep(rand() % 128);
                sem_post(mutex);
                sleep(rand() % 128);
                sem_post(empty);
                sleep(rand() % 128);
        }
}
```

```c

void test_producer_consumer()
{
        sem_t mutex, full, empty;
        sem_init(&mutex, 1);
        sem_init(&full, 0);
        sem_init(&empty, N);
        int id;
        for (int i = 0; i < N - 1; ++i)
        {
                int ret = fork();
                if (ret == -1)
                {
                        printf("fork error: %d\n", i);
                        exit();
                }
                if (ret == 0) // child
                {
                        id = get_pid();
                        producer(id, &mutex, &full, &empty);
                }
        }
        id = get_pid();
        consumer(&mutex, &full, &empty);
}
```

2.3.3 读写者问题

这个问题和前面两个问题不一样，它需要有一个 reader_count 的整数变量，并且需要在进程间共享。这是一个与信号量完全不同的要求，必须添加一个全新的系统调用来实现。其实这在 os 中应该是进程间共享内存通信，这里做一个简化的实现——在内核中开辟一块内存，通过系统调用对这块内存进行读写。将系统调用命名为 SYS_SHM(shared memory)，提供 read 和 write 接口。

```c
// in kernel
void syscallShm(struct StackFrame *sf)
```

```
{
        switch (sf->ecx)
        {
        case SHM_WRITE:
                shm = sf->edx;
                break;
        case SHM_READ:
                sf->eax = shm;
                break;
        default:
                break;
        }
}
// in user lib
uint32_t shm_read()
{
        return syscall(SYS_SHM, SHM_READ, 0, 0, 0, 0);
}
int shm_write(uint32_t value)
{
        return syscall(SYS_SHM, SHM_WRITE, value, 0, 0, 0);
}
```

通过 fork 函数产生 5 个子进程，前 2 个子进程和父进程调用 reader 过程，后 3 个子进程调用 writer 过程。

```
void writer(sem_t *writemutex)
{
        int id = get_pid() - 4;
        while (1)
        {
                sem_wait(writemutex);
                printf("Writer %d: write\n", id);
                sleep(rand() % 128);
                sem_post(writemutex);
                sleep(rand() % 128);
        }
}
void reader(sem_t *writemutex, sem_t *countmutex)
```

```
{
        int id = get_pid() - 1;
        int rcount;
        while (1)
        {
                sem_wait(countmutex);
                sleep(rand() % 128);
                rcount = shm_read();
                if (rcount == 0)
                        sem_wait(writemutex);
                sleep(rand() % 128);
                rcount = shm_read();
                ++rcount;
                shm_write(rcount);
                sleep(rand() % 128);
                sem_post(countmutex);
                printf("Reader %d: read, total %d reader\n", id,
                    rcount);
                sleep(rand() % 128);
                sem_wait(countmutex);
                rcount = shm_read();
                --rcount;
                shm_write(rcount);
                sleep(rand() % 128);
                rcount = shm_read();
                if (rcount == 0)
                        sem_post(writemutex);
                sleep(rand() % 128);
                sem_post(countmutex);
                sleep(rand() % 128);
        }
}

void test_reader_writer()
{
        sem_t writemutex, countmutex;
        sem_init(&writemutex, 1);
        sem_init(&countmutex, 1);
```

```
        shm_write(0);
        int ret = 1;
        for (int i = 0; i < 5; ++i)
                if (ret > 0)
                        ret = fork();
        int id = get_pid();
        if (id < 4)
                reader(&writemutex, &countmutex);
        else if (id < 7)
                writer(&writemutex);
}
```

3 实验结果

3.1 scanf 和信号量

scanf 可以正确接受输入，信号量调用正常。

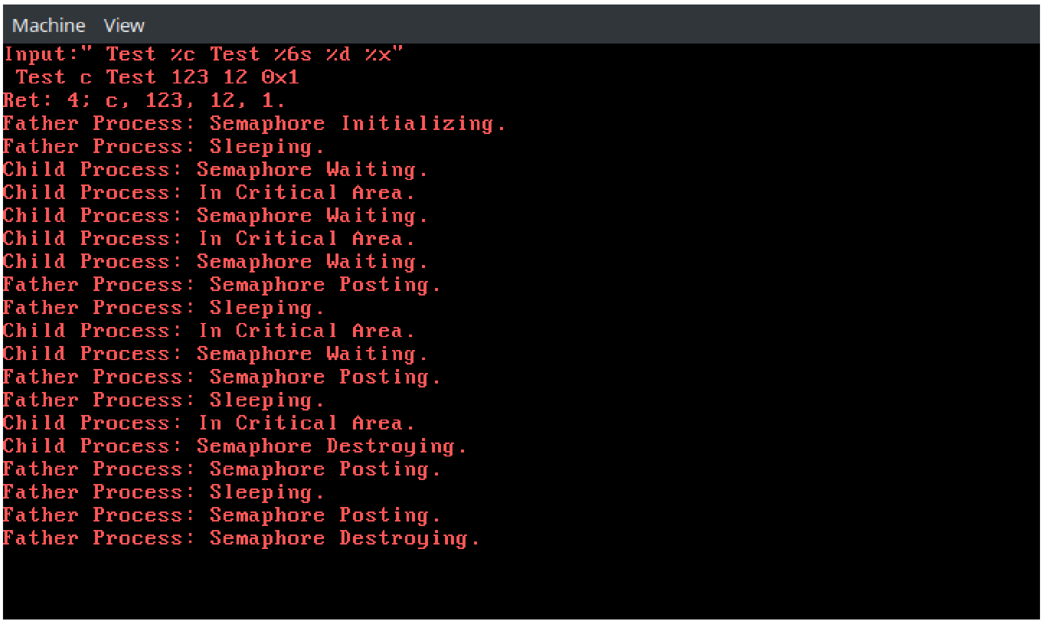

图 1: scanf 和信号量运行结果

3.2 哲学家问题

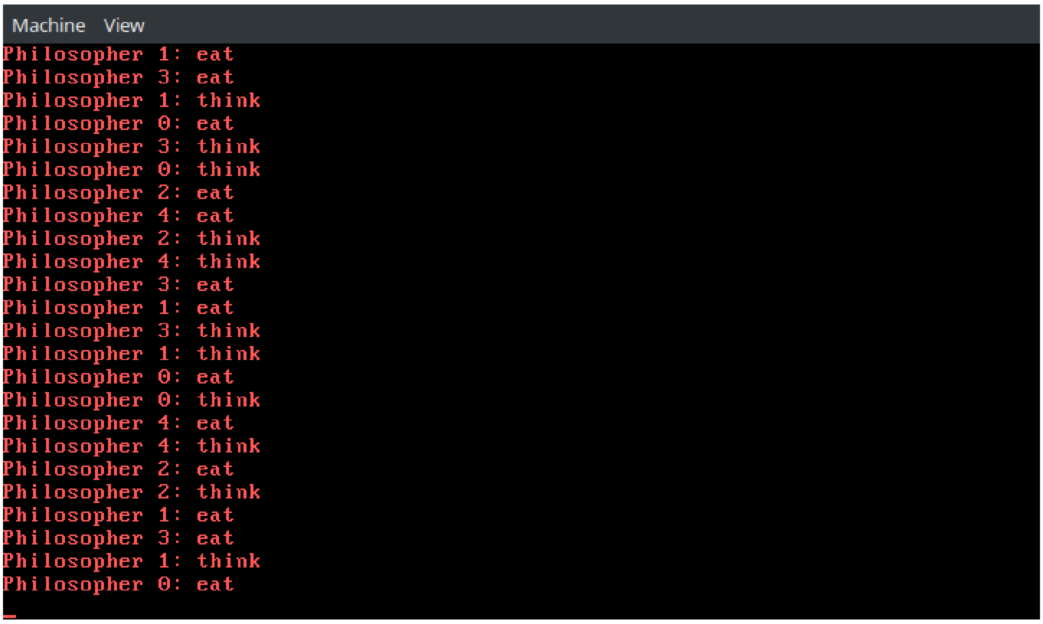

图 2: 哲学家问题结果

3.3 生产消费问题

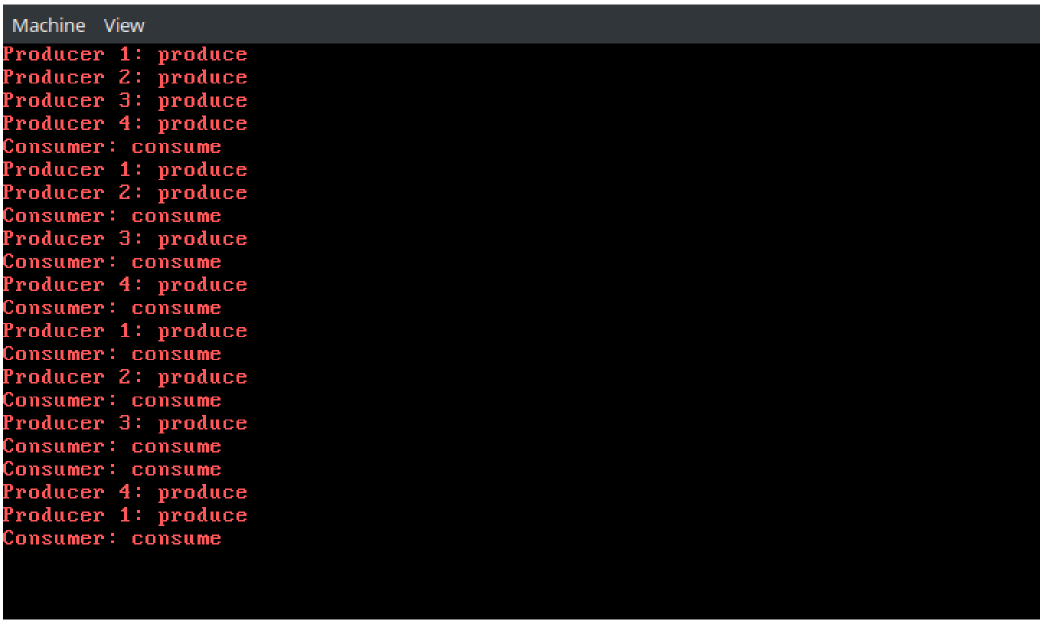

图 3: 生产消费问题运行结果

3.4 读写者问题

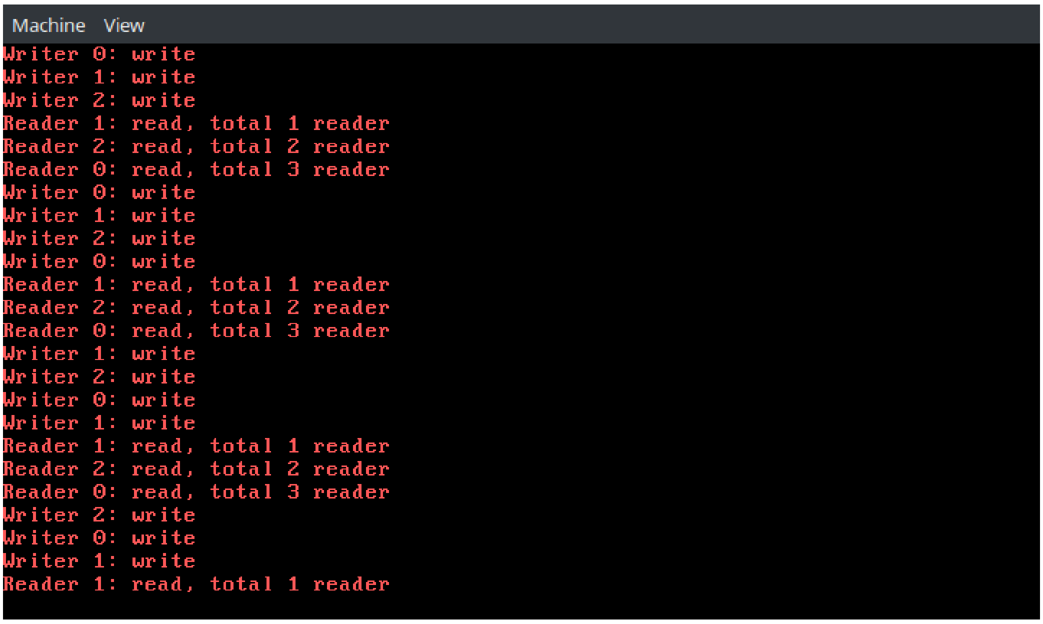

图 4: 读写者问题运行结果

4 问题与思考

1. 在命令行中运行 qemu-system-i386 os.img，窗口显示”no bootable device”。原因可能是 gcc 版本问题。安装 gcc6 并编译可成功运行。另外，还有一个解决方案是加一个编译参数-fno-asynchronous-unwind-tables，经过尝试也可以成功。查询相关资料知，”This option determines whether unwind information is precise at an instruction boundary or at a call boundary. If -fno-asynchronous-unwind-tables is specified, the unwind table is precise at call boundaries only.” 新版本的 gcc 默认行为是”-fasynchronous-unwind-tables”，而旧版的 gcc 默认”-fno-asynchronous-unwind-tables”。猜想 unwind information 的位置不同将影响加载过程，而 qemu 模拟的是老式的硬件，可能会产生不匹配。

2. 一开始写哲学家问题时总是出现 fork 返回-1 的情况，经过很长时间的

检查才发现框架代码默认只支持 4 个用户进程。修改 NR_SEGMENT 可以改变最大进程数，才能正确运行。

5 建议

1. 建议将官方的实验环境升级到 20.04LTS 版本，这样不容易产生 gcc 版本问题，可以节省很多时间;

2. 建议将 memery.h 中的 NR_SEGMENT 直接修改为 20。