



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	2
2	实验思路 and 过程	3
2.1	scanf	3
2.2	信号量	5
2.3	进程同步与通信	6
2.3.1	哲学家问题	7
2.3.2	生产者消费者问题	9
2.3.3	读写者问题	10
3	实验结果	13
3.1	scanf 和信号量	13
3.2	哲学家问题	14
3.3	生产消费问题	15
3.4	读写者问题	15
4	问题与思考	15
5	建议	16

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）。

```
1 int uEntry(void)
2 {
3     // For lab4.1
4     test_scanf();
5
6     // For lab4.2
7     test_sem();
8
9     // For lab4.3
10    //! note that this function is non-stopping
11    test_philosopher();
12
13    // Optional
14    //! note that these functions are non-stopping
15    test_producer_consumer();
16    test_reader_writer();
17
18    return 0;
19 }
```

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 and 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。

```
1 void keyboardHandle(struct StackFrame *sf)
2 {
3     uint32_t keyCode = getKeyCode();
4     if (keyCode == 0) // illegal keyCode
5         return;
```

```

6
7     keyBuffer[bufferTail] = keyCode;
8     bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;
9
10    assert(dev[STD_IN].value == -1 || dev[STD_IN].value ==
        0);
11    if (dev[STD_IN].value < 0)
12    { // with process blocked
13        // TODO: deal with blocked situation
14        ProcessTable *pt = pop_dev(STD_IN);
15        ++dev[STD_IN].value;
16        pt->state = STATE_RUNNABLE;
17    }
18
19    return;
20 }

```

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。

```

1 void syscallReadStdIn(struct StackFrame *sf)
2 {
3     if (dev[STD_IN].value < 0)
4     {
5         sf->eax = -1;
6         return;
7     }
8     --dev[STD_IN].value;

```

```

9      push_dev(current, STD_IN);
10     pcb[current].state = STATE_BLOCKED;
11     asm volatile("int $0x20");
12     int sel = sf->ds;
13     char *str = (char *)sf->edx;
14     int size = sf->ebx;
15     int i = 0;
16     char character;
17     asm volatile("movw %0, %%es" ::"m"(sel));
18     while (i < size - 1 && bufferHead != bufferTail)
19     {
20         character = getChar(keyBuffer[bufferHead]);
21         bufferHead = (bufferHead + 1) %
22             MAX_KEYBUFFER_SIZE;
23         if (character != 0)
24         {
25             stdout_char(character);
26             asm volatile("movb %0, %%es:(%1)" ::"r"(
27                 character), "r"(str + i));
28             ++i;
29         }
30     }
31     asm volatile("movb $0, %%es:(%0)" ::"r"(str + i));
32     sf->eax = i;
33 }

```

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。

```

1 void syscallSemWait(struct StackFrame *sf)
2 {
3     int i = (int)sf->edx;
4     if (i < 0 || i >= MAX_SEM_NUM)
5     {
6         pcb[current].regs.eax = -1;
7         return;
8     }
9 }

```

```

8     }
9     if (--sem[i].value < 0)
10    {
11        push_sem(current, i);
12        // ++pcb[current].blocked_sems;
13        pcb[current].state = STATE_BLOCKED;
14        asm volatile("int $0x20");
15    }
16    sf->eax = 0;
17 }

```

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。

```

1 void syscallSemPost(struct StackFrame *sf)
2 {
3     int i = (int)sf->edx;
4     if (i < 0 || i >= MAX_SEM_NUM)
5     {
6         pcb[current].regs.eax = -1;
7         return;
8     }
9     if (++sem[i].value <= 0)
10    {
11        ProcessTable *pt = pop_sem(i);
12        // if (!--pt->blocked_sems)
13        pt->state = STATE_RUNNABLE;
14    }
15    sf->eax = 0;
16 }

```

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。

```

1 uint32_t next = 0;
2 uint32_t rand()
3 {
4     return next = next * 1103515245 + 12345;
5 }

```

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 memory.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。

```

1 #define N 5 // number of philosopher
2 void philosopher(int i, sem_t *forks)
3 {
4     --i; // pid start at 1
5     while (1)
6     {
7         if (i % 2 == 0)
8         {
9             sem_wait(&forks[i]);
10            sleep(rand() % 128);
11            sem_wait(&forks[(i + 1) % N]);
12        }
13        else
14        {
15            sem_wait(&forks[(i + 1) % N]);
16            sleep(rand() % 128);
17            sem_wait(&forks[i]);

```



```

18         }
19         printf("Philosopher %d: eat\n", i);
20         sleep(rand() % 128 + 64); // eat
21         printf("Philosopher %d: think\n", i);
22         sem_post(&forks[i]);
23         sleep(rand() % 128);
24         sem_post(&forks[(i + 1) % N]);
25         sleep(rand() % 128 + 64); // think
26     }
27 }
28
29 void test_philosopher()
30 {
31     sem_t forks[N];
32     int id;
33     for (int i = 0; i < N; ++i)
34         sem_init(&forks[i], 1);
35     for (int i = 0; i < N - 1; ++i)
36     {
37         int ret = fork();
38         if (ret == -1)
39         {
40             printf("fork error: %d\n", i);
41             exit();
42         }
43         if (ret == 0) // child
44         {
45             id = get_pid();
46             philosopher(id, forks);
47         }
48     }
49     id = get_pid();
50     philosopher(id, forks);
51 }

```

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。

```
1 void producer(int id, sem_t *mutex, sem_t *full, sem_t *empty)
2 {
3     --id; // pid start at 1
4     while (1)
5     {
6         sem_wait(empty);
7         sleep(rand() % 128);
8         sem_wait(mutex);
9         sleep(rand() % 128);
10        printf("Producer %d: produce\n", id);
11        sleep(rand() % 128);
12        sem_post(mutex);
13        sleep(rand() % 128);
14        sem_post(full);
15        sleep(rand() % 128);
16    }
17 }
18
19 void consumer(sem_t *mutex, sem_t *full, sem_t *empty)
20 {
21     while (1)
22     {
23         sem_wait(full);
24         sleep(rand() % 128);
25         sem_wait(mutex);
26         sleep(rand() % 128);
27         printf("Consumer: consume\n");
28         sleep(rand() % 128);
29         sem_post(mutex);
30         sleep(rand() % 128);
31         sem_post(empty);
32         sleep(rand() % 128);
33     }
34 }
```

```

35
36 void test_producer_consumer()
37 {
38     sem_t mutex, full, empty;
39     sem_init(&mutex, 1);
40     sem_init(&full, 0);
41     sem_init(&empty, N);
42     int id;
43     for (int i = 0; i < N - 1; ++i)
44     {
45         int ret = fork();
46         if (ret == -1)
47         {
48             printf("fork error: %d\n", i);
49             exit();
50         }
51         if (ret == 0) // child
52         {
53             id = get_pid();
54             producer(id, &mutex, &full, &empty);
55         }
56     }
57     id = get_pid();
58     consumer(&mutex, &full, &empty);
59 }

```

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。

```

1 // in kernel
2 void syscallShm(struct StackFrame *sf)

```

```

3 {
4     switch (sf->ecx)
5     {
6         case SHM_WRITE:
7             shm = sf->edx;
8             break;
9         case SHM_READ:
10            sf->eax = shm;
11            break;
12        default:
13            break;
14    }
15 }
16 // in user lib
17 uint32_t shm_read()
18 {
19     return syscall(SYS_SHM, SHM_READ, 0, 0, 0, 0);
20 }
21 int shm_write(uint32_t value)
22 {
23     return syscall(SYS_SHM, SHM_WRITE, value, 0, 0, 0);
24 }

```

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。

```

1 void writer(sem_t *writemutex)
2 {
3     int id = get_pid() - 4;
4     while (1)
5     {
6         sem_wait(writemutex);
7         printf("Writer %d: write\n", id);
8         sleep(rand() % 128);
9         sem_post(writemutex);
10        sleep(rand() % 128);
11    }
12 }
13 void reader(sem_t *writemutex, sem_t *countmutex)

```

```

14 {
15     int id = get_pid() - 1;
16     int rcount;
17     while (1)
18     {
19         sem_wait(countmutex);
20         sleep(rand() % 128);
21         rcount = shm_read();
22         if (rcount == 0)
23             sem_wait(writemutex);
24         sleep(rand() % 128);
25         rcount = shm_read();
26         ++rcount;
27         shm_write(rcount);
28         sleep(rand() % 128);
29         sem_post(countmutex);
30         printf("Reader %d: read, total %d reader\n", id,
31             rcount);
32         sleep(rand() % 128);
33         sem_wait(countmutex);
34         rcount = shm_read();
35         --rcount;
36         shm_write(rcount);
37         sleep(rand() % 128);
38         rcount = shm_read();
39         if (rcount == 0)
40             sem_post(writemutex);
41         sleep(rand() % 128);
42         sem_post(countmutex);
43         sleep(rand() % 128);
44     }
45 }
46 void test_reader_writer()
47 {
48     sem_t writemutex, countmutex;
49     sem_init(&writemutex, 1);
50     sem_init(&countmutex, 1);

```

```

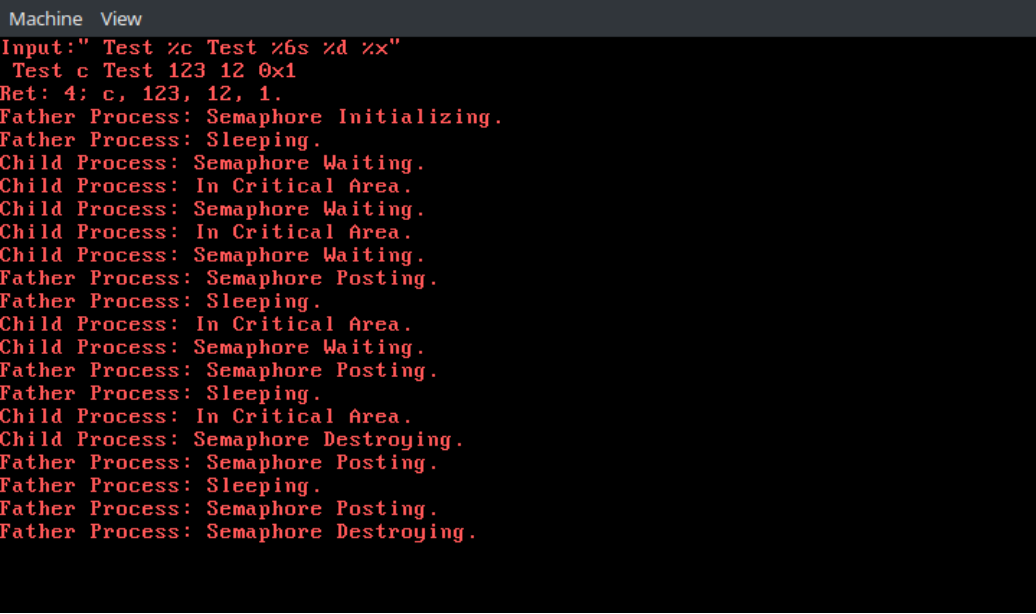
51     shm_write(0);
52     int ret = 1;
53     for (int i = 0; i < 5; ++i)
54         if (ret > 0)
55             ret = fork();
56     int id = get_pid();
57     if (id < 4)
58         reader(&writemutex, &countmutex);
59     else if (id < 7)
60         writer(&writemutex);
61 }

```

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。



```

Machine View
Input: " Test %c Test %6s %d %x"
Test c Test 123 12 0x1
Ret: 4: c, 123, 12, 1.
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.

```

图 1: scanf 和信号量运行结果

3.2 哲学家问题

```
Machine View
Philosopher 1: eat
Philosopher 3: eat
Philosopher 1: think
Philosopher 0: eat
Philosopher 3: think
Philosopher 0: think
Philosopher 2: eat
Philosopher 4: eat
Philosopher 2: think
Philosopher 4: think
Philosopher 3: eat
Philosopher 1: eat
Philosopher 3: think
Philosopher 1: think
Philosopher 0: eat
Philosopher 0: think
Philosopher 4: eat
Philosopher 4: think
Philosopher 2: eat
Philosopher 2: think
Philosopher 1: eat
Philosopher 3: eat
Philosopher 1: think
Philosopher 0: eat
```

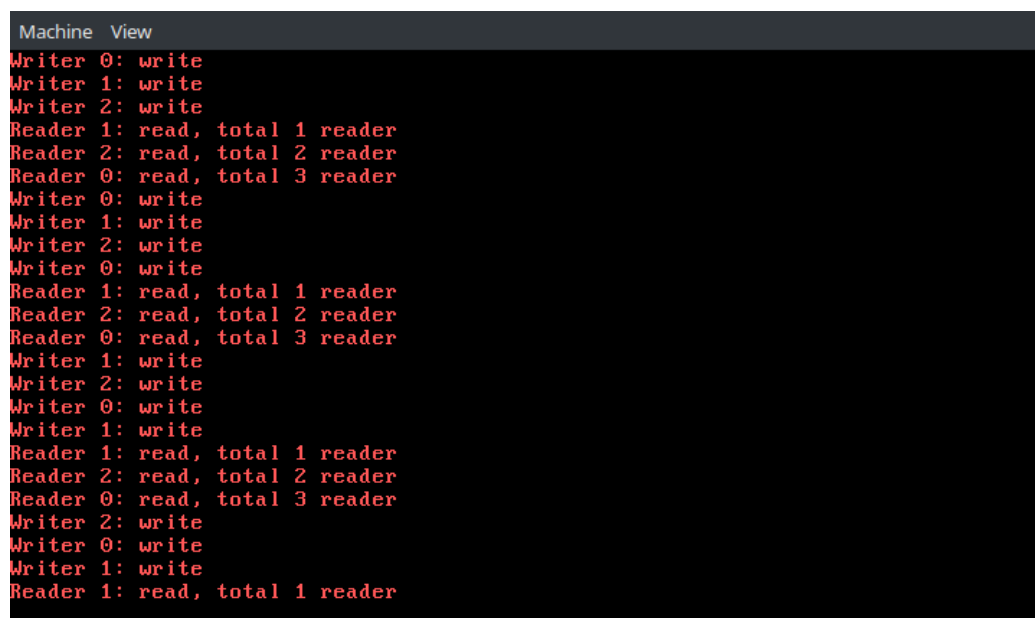
图 2: 哲学家问题结果

3.3 生产消费问题

```
Machine  View
Producer 1: produce
Producer 2: produce
Producer 3: produce
Producer 4: produce
Consumer: consume
Producer 1: produce
Producer 2: produce
Consumer: consume
Producer 3: produce
Consumer: consume
Producer 4: produce
Consumer: consume
Producer 1: produce
Consumer: consume
Producer 2: produce
Consumer: consume
Producer 3: produce
Consumer: consume
Consumer: consume
Producer 4: produce
Producer 1: produce
Consumer: consume
```

图 3: 生产消费问题运行结果

3.4 读写者问题



```
Machine View
Writer 0: write
Writer 1: write
Writer 2: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 0: write
Writer 1: write
Writer 2: write
Writer 0: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 1: write
Writer 2: write
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
Reader 2: read, total 2 reader
Reader 0: read, total 3 reader
Writer 2: write
Writer 0: write
Writer 1: write
Reader 1: read, total 1 reader
```

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 `memory.h` 中的 `NR_SEGMENT` 直接修改为 20。