

《操作系统》lab 4实验报告

姓名	学号	邮箱	日期
薛飞阳	181860121	502126785@qq.com	2020/4/23

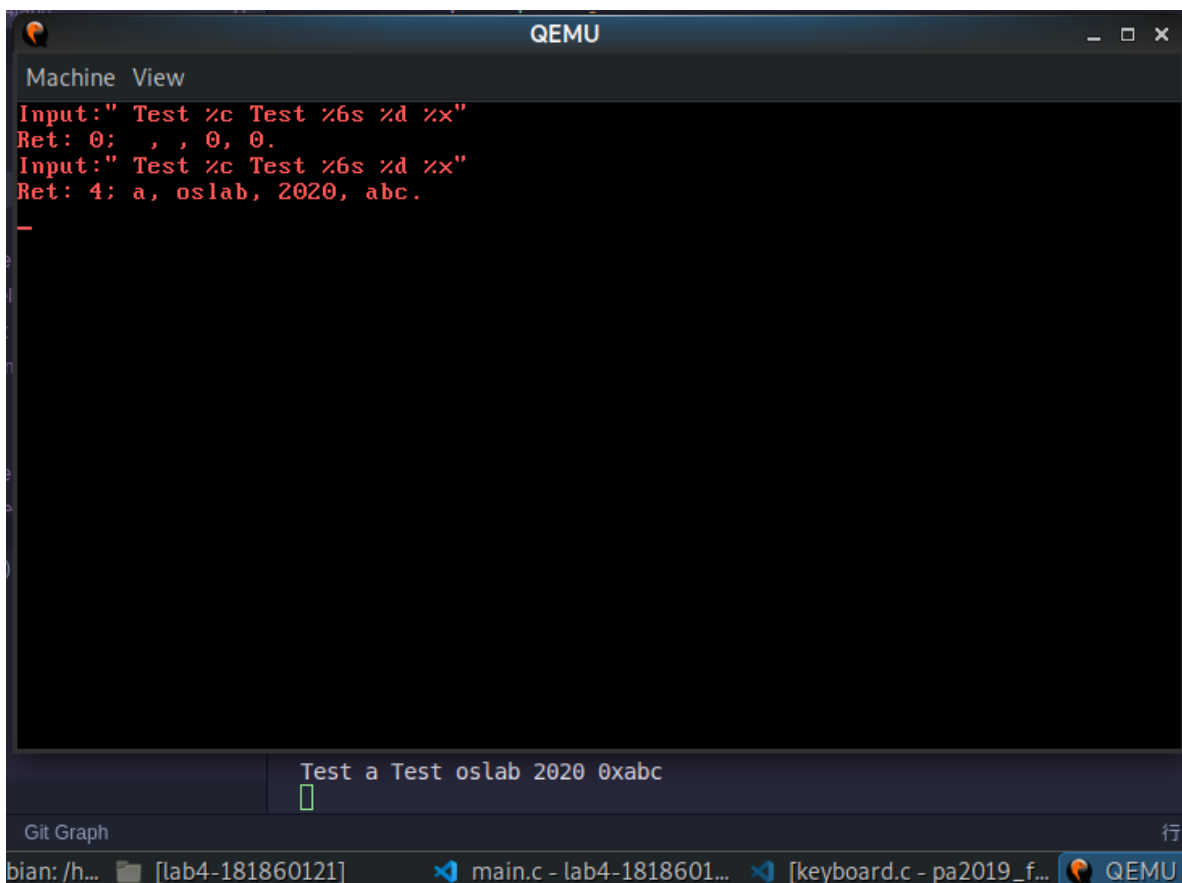
1. 实验进度

我完成了全部必做内容，以及选做中的随机函数

2. 实验结果

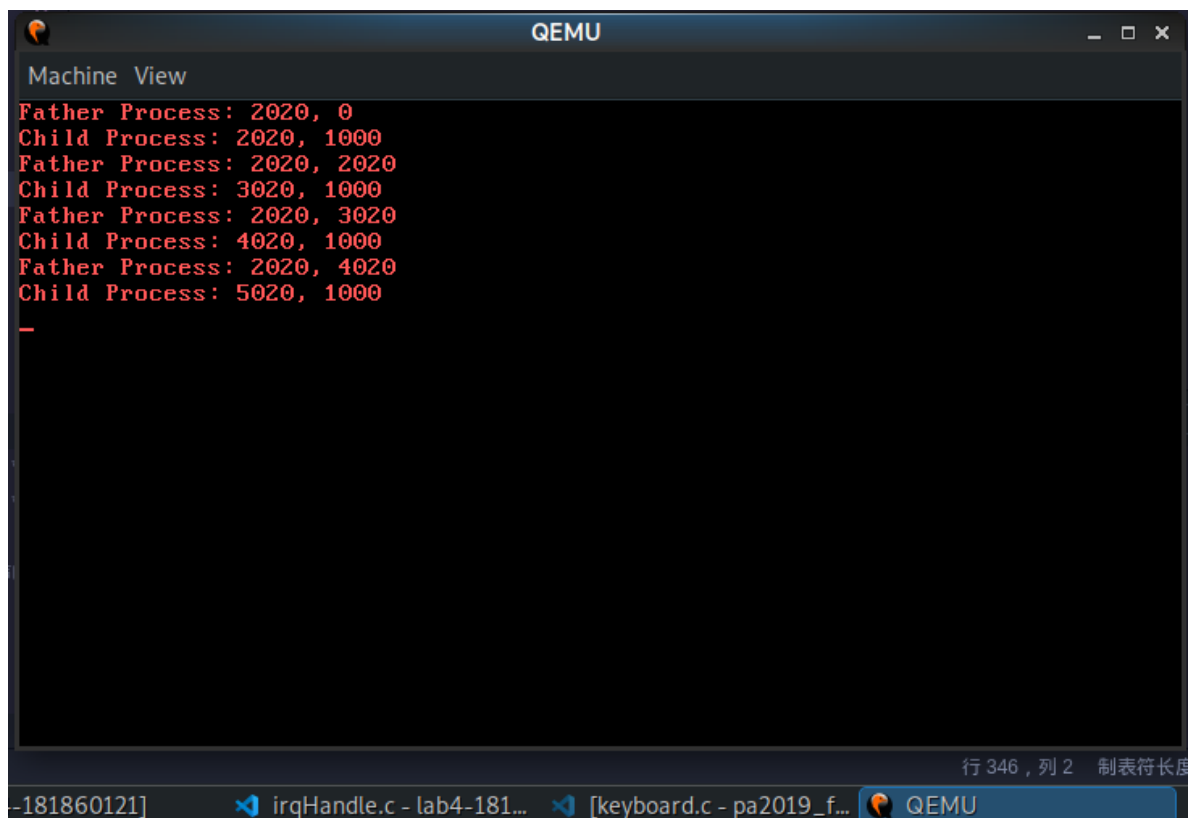
//截图为开启了随机中断时间的运行结果

2.1. 格式化输入scanf



```
QEMU
Machine View
Input: " Test %c Test %6s %d %x"
Ret: 0: , , 0, 0.
Input: " Test %c Test %6s %d %x"
Ret: 4: a, oslab, 2020, abc.
Test a Test oslab 2020 0xabc
```

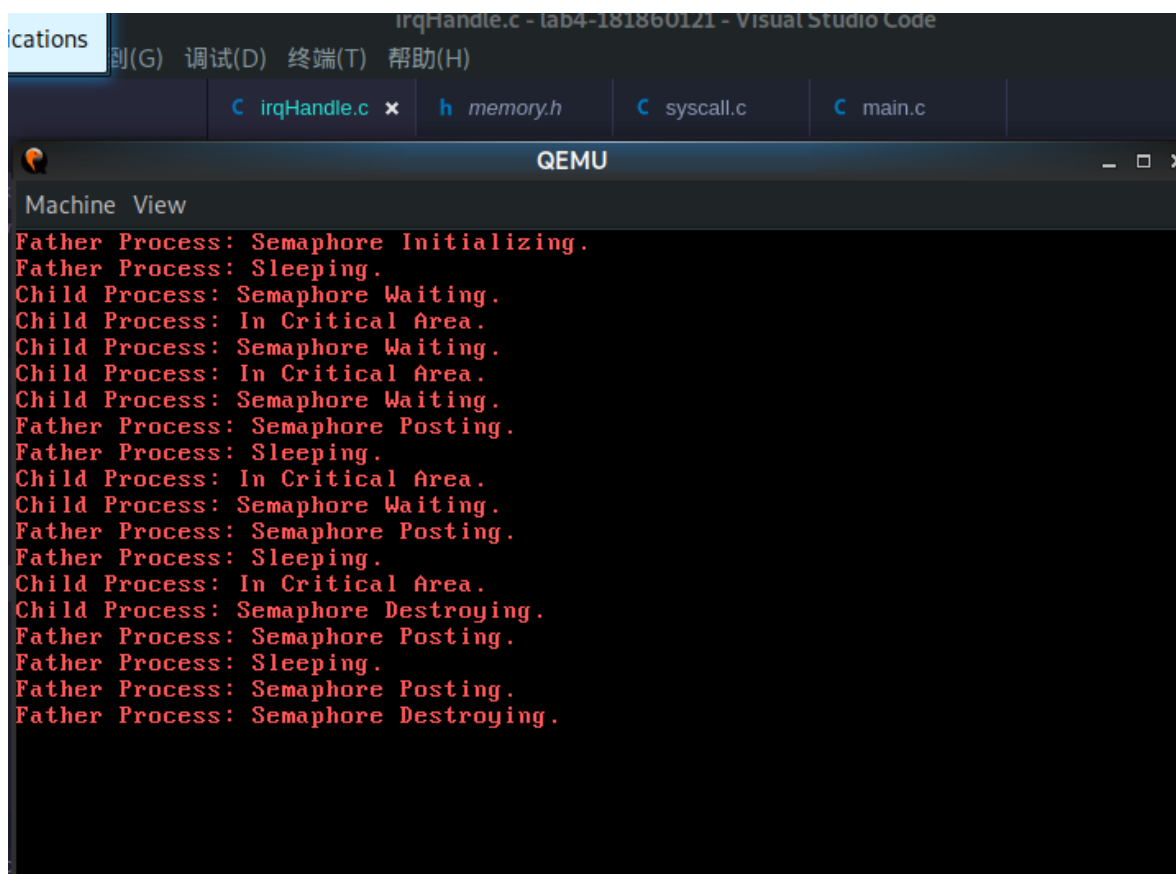
2.2. 进程通信ShMem



The screenshot shows a QEMU terminal window titled "Machine View". The output displays a sequence of process creation and execution: a father process (PID 2020) creates child processes (PIDs 1000, 2020, 3020, 4020, 5020) in a nested fashion. The status bar at the bottom indicates the current line is 346, column 2, and the file being edited is "irqHandle.c".

```
Machine View
Father Process: 2020, 0
Child Process: 2020, 1000
Father Process: 2020, 2020
Child Process: 3020, 1000
Father Process: 2020, 3020
Child Process: 4020, 1000
Father Process: 2020, 4020
Child Process: 5020, 1000
-
行 346 , 列 2 制表符长度
-181860121] irqHandle.c - lab4-181... [keyboard.c - pa2019_f... QEMU
```

2.3. 信号量调用Sem

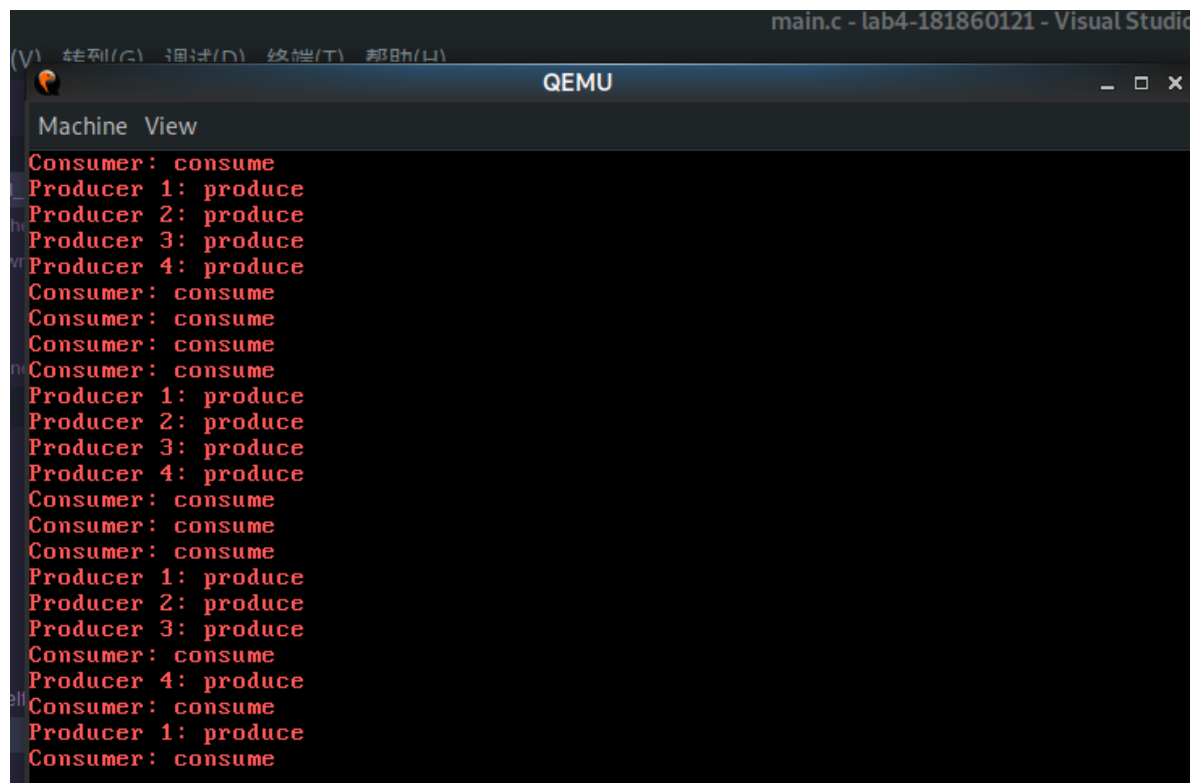


The screenshot shows a Visual Studio Code editor with a QEMU terminal window titled "Machine View". The output displays the execution of a semaphore: the father process initializes and sleeps on the semaphore, while child processes wait, enter critical areas, and then the father process posts the semaphore. The status bar at the bottom indicates the current line is 346, column 2, and the file being edited is "irqHandle.c".

```
Machine View
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.
行 346 , 列 2 制表符长度
-181860121] irqHandle.c - lab4-181860121 - Visual Studio Code
```

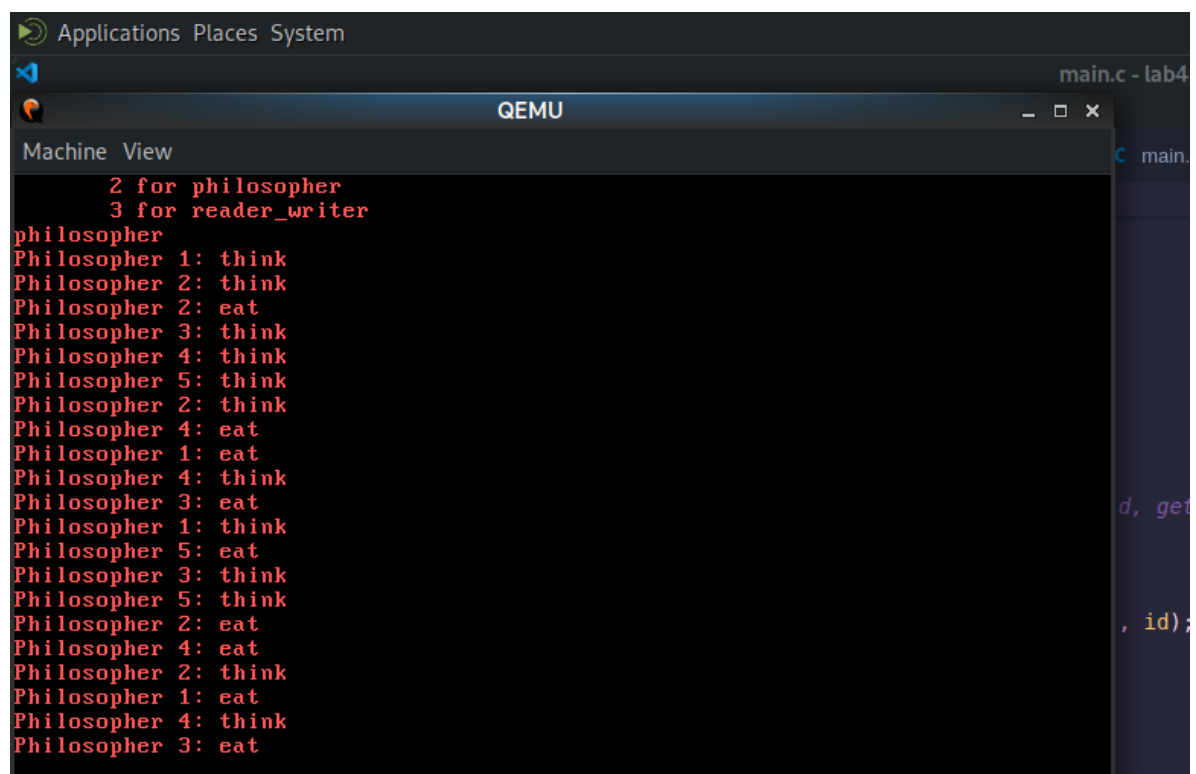
2.4. 进程同步问题

2.4.1. 生产者-消费者



```
main.c - lab4-181860121 - Visual Studio
(V) 转到(G) 调试(D) 终端(T) 帮助(H)
QEMU
Machine View
Consumer: consume
Producer 1: produce
Producer 2: produce
Producer 3: produce
Producer 4: produce
Consumer: consume
Consumer: consume
Consumer: consume
Producer 1: produce
Producer 2: produce
Producer 3: produce
Producer 4: produce
Consumer: consume
Consumer: consume
Consumer: consume
Producer 1: produce
Producer 2: produce
Producer 3: produce
Consumer: consume
Producer 4: produce
Consumer: consume
Producer 1: produce
Consumer: consume
```

2.4.2. 哲学家就餐



```
Applications Places System
main.c - lab4
QEMU
Machine View
2 for philosopher
3 for reader_writer
philosopher
Philosopher 1: think
Philosopher 2: think
Philosopher 2: eat
Philosopher 3: think
Philosopher 4: think
Philosopher 5: think
Philosopher 2: think
Philosopher 4: eat
Philosopher 1: eat
Philosopher 4: think
Philosopher 3: eat
Philosopher 1: think
Philosopher 5: eat
Philosopher 3: think
Philosopher 5: think
Philosopher 2: eat
Philosopher 4: eat
Philosopher 2: think
Philosopher 1: eat
Philosopher 4: think
Philosopher 3: eat
```

2.4.3. 读者-写者

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

3 reader - 3 writer

3. 实验修改的代码

3.1. 实现格式化输入函数

修改位置: `lab4/kernel/kernel/irqHandle.c`

在 `irqHandle.c` 中, 完成 `keyboardHandle()` 和 `syscallReadStdIn()` 函数, 为函数增加格式化输入功能。

3.1.1. `keyboardHandle()`

`keyboardHandle()` 完成处理键盘的任务, 要求其读取键盘码并将其填入缓冲区。完成后, 唤醒阻塞在 `dev[STD_IN]` 上的一个进程。由于格式化输入需要终止符, 而通过键盘输入一般以 `\n` 作为终止, 故可以在该函数中添加判断, 直到读到 `\n` 之后, 才开始唤醒阻塞在 `dev[STD_IN]` 上的一个进程, 这样就可以实现 `scanf` 的终止。

需要注意的是判断缓冲区是否已满, 以及 `bufferTail` 的取余增加。该函数的处理流程如下:

```
if buffer not full:
    #Put in buffer
    bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE
if dev[STD_IN].value < 0 and get '\n':
    dev[STD_IN].value += 1
    #Awake Process
```

具体代码部分不再展示。为方便调试，在缓冲区已满时调用 `putString()` 输出相关信息。

3.1.2. `syscallReadStdIn()`

`syscallReadStdIn()` 会在 `dev[STD_IN].value == 0` 时阻塞当前进程，然后等待唤醒。唤醒后读取 `buffer` 中的所有数据，若 `value < 0` 则失败，返回-1。由于之前的调度对于BLOCKED状态的进程有时间片的设定，当一定时间后自动唤醒，为了防止这种情况，手动将其`sleepTime`设为-1（之后唤醒也要修改其值为0），然后，调用 `int $0x20` 进行进程的切换。

观察 `scanf()` 函数，发现其需要将读入的值放入 `tf->edx` 指向的 `buffer`，同时指明 `buffer` 的最大值为 `tf->ebx`。当进程被唤醒，即意味着 `keyboardHandle()` 执行完毕，数据已经全部读入缓冲区，这时候，就需要依次将数据转换为字符，放入 `buffer` 中。参照 `printf()`，由于此时处于内核态，还需要借用 `tf->ds`，才能正确地将字符放入数组中。在全部读取完后，按照定义，还需要在字符串的末尾放上 `'\0'`。所以实际只能最多存放 `size - 1` 个有效字符。这部分代码如下所示：

```
int sel = tf->ds;
char *str = (char *)tf->edx;
int size = tf->ebx;
char character;
int i = 0;
asm volatile("movw %0, %%es" :: "m"(sel));
while (bufferHead != bufferTail)
{
    character = getChar(keyBuffer[bufferHead]);
    bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
    if (character == 0)
        continue;
    asm volatile("movb %0, %%es:(%1)" :: "r"(character), "r"(str + i));
    i++;
    putChar(character);
    if (i == size - 1)
    {
        //buffer overflow
        putString("syscallReadStdIn:Input overflow!");
        break;
    }
}
character = 0;
asm volatile("movb %0, %%es:(%1)" :: "r"(character), "r"(str + i)); //add \0 to the end
pcb[current].regs.eax = i;
```

同样地，在 `buffer` 已满时调用 `putString()` 输出相关信息。

3.2. 实现进程通信

修改位置：`lab4/kernel/kernel/irqHandle.c`

这一部分比较简单，填写在 `irqHandle.c` 中的 `syscallWriteShMem()` 和 `syscallReadShMem()` 函数即可。其处理流程基本可以参照标准输入输出函数，只不过输入输出位置换为 `ShMem`。查看 `syscall.c` 中的 `Read()` 和 `Write()` 函数，发现参数包括：

```
buffer:tf->edx //指明要写/读的字符串首地址
size:tf->ebx   //buffer的最大大小
index:tf->esi  //在共享内存中读/写的起始位置
```

以 `syscallWriteShMem()` 为例，该函数将数据写入共享内存中，其执行过程如下所示：

```
int sel = tf->ds;
uint8_t *str = (uint8_t *)tf->edx;
int size = tf->ebx;
int index = tf->esi;
int i = 0;
asm volatile("movw %0, %%es" :: "m"(sel));
uint8_t c;
while (i < size)
{
    asm volatile("movb %%es:(%1), %0"
                  : "=r"(c)
                  : "r"(str + i));
    shMem[i + index] = c;
    i++;
    if (i + index == MAX_SHMEM_SIZE)
    {
        //shmem overflow
        putString("syscallWriteShMem:ShMem overflow!");
        i = -1;
        break;
    }
}
pcb[current].regs.eax = i;
return;
```

3.3. 实现信号量

修改位置： `lab4/kernel/kernel/irqHandle.c`

这一部分较为简单，需要完成4个函数，执行信号量的 `init`、`wait`、`post`、`destroy` 四个功能。其具体如何实现参考下图即可：

sem_init

`sem_init` 系统调用用于初始化信号量，其中参数 `value` 用于指定信号量的初始值，初始化成功则返回 0，指针 `sem` 指向初始化成功的信号量，否则返回 -1

```
int sem_init(sem_t *sem, uint32_t value);
```

sem_post

`sem_post` 系统调用对应信号量的 V 操作，其使得 `sem` 指向的信号量的 `value` 增一，若 `value` 取值不大于 0，则释放一个阻塞在该信号量上进程（即将该进程设置为就绪态），若操作成功则返回 0，否则返回 -1

```
int sem_post(sem_t *sem);
```

sem_wait

`sem_wait` 系统调用对应信号量的 P 操作，其使得 `sem` 指向的信号量的 `value` 减一，若 `value` 取值小于 0，则阻塞自身，否则进程继续执行，若操作成功则返回 0，否则返回 -1

```
int sem_wait(sem_t *sem);
```

sem_destroy

`sem_destroy` 系统调用用于销毁 `sem` 指向的信号量，销毁成功则返回 0，否则返回 -1，若尚有进程阻塞在该信号量上，可带来未知错误

```
int sem_destroy(sem_t *sem);
```

需要注意的是，对于 wait、post、destroy 操作，其操作的下标来自于 `tf->edx`，而这个下标是在 `init` 中确定的，方法为：寻找到第一个 state 为 0 的 `sem` 作为下标，然后将该下标作为返回值返回，如下所示：

```
int value = (int)tf->edx;
int i = 0;
for (; i < MAX_SEM_NUM; i++)
{
    if (sem[i].state == 0)
        break;
}
if (i == MAX_SEM_NUM)
{
    putString("syscallSemInit:No useable sem!");
    pcb[current].regs.eax = -1;
    return;
}
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;
```

3.4. 进程同步问题

3.4.1. 随机函数（选做）

修改位置：lab4/lib/syscall.c & lab4/lib/lib.h

由于实现了随机函数，首先介绍这一部分的内容。

观察用户文件（各个main.c），发现其使用的系统函数的声明和定义分别位于lib.h以及syscall.c中，故这部分在此实现。

网上查阅C语言自带的随机数生成函数rand()和srand(int seed)，前者的功能是根据当前seed通过一定方法计算出，后者的功能是根据传入值直接设置一个种子。仿照这个概念，自定义函数getrand(int max)和setrand(int seed)，为了方便使用，前者额外传入的max参数设置返回的最大值，两个函数的实现如下所示：

```
static unsigned long myseed = 1;

int getrand(int max)
{
    myseed = myseed * 1103515245 + 12345;
    return ((unsigned)(myseed / 65536) % 32768) % max;
}

void setrand(int seed)
{
    myseed = seed;
}
```

然后在lib.h中填写声明即可。这样，如果在之后需要生成一个0到128的随机数，调用getrand(128)即可。

3.4.2. 生产者-消费者

修改位置：lab4/bounded_buffer/main.c

本问题需要生成4个生产者和一个消费者，因此，需要在main函数里fork出5个子进程，对于第一个执行消费者功能，后面的执行生产者功能，如下所示：

```
for (int i = 0; i < 5; i++)
{
    int ret = fork();
    if (ret == 0)
    {
        setrand(i + 1);
        //make 1 consumer and 4 producer
        if (i == 0)
```



```

        consumer();
    else
    {
        producer(i);
    }
    break;
}
}

```

`setrand()` 为每个子进程设立了独特的种子，增加了随机函数的随机性。

对于本问题，我们需要3个信号量，`mutex`用于实现互斥，生产者和消费者只能有一个在对缓冲区进行操作；`house_surplus`指明仓库的剩余大小，当仓库满时（剩余为0）生产者将被阻塞；`product_surplus`指明产品剩余数量，当产品空时消费者将被阻塞，其初始化在 `main()` 中完成，如下所示：

```

sem_init(&mutex, 1);
sem_init(&house_surplus, MAX_PRODUCE_BUFFER);
sem_init(&product_surplus, 0);

```

其中`MAX_PRODUCE_BUFFER`表示缓冲区的最大值，这里设为10。

根据上述描述，我们可以规划出消费者和生产者函数分别需要完成的任务，简单来说，消费者每次先让产品数量--，再让仓库大小++，然后开始**消费**；生产者则先开始**生产**，然后让仓库大小--，再让产品数量++。具体操作如下：

```

void consumer()
{
    //printf("Consumer: pid:%d\n", getpid());
    int rand;
    while (1)
    {
        sem_wait(&product_surplus); //wait until have product to consume
        sem_wait(&mutex);
        step_control++;
        if (step_control >= MAX_STEP)
            break;
        rand = (getrand(64));
        sem_post(&mutex);
        sem_post(&house_surplus);    //consumed, house++
        sleep(rand);
        printf("Consumer: consume\n");
    }
}

void producer(int id)
{
    //printf("Producer %d: pid:%d\n", id, getpid());
    int rand = (getrand(96) + 64);
    while (1)
    {
        printf("Producer %d: produce\n", id);
        sleep(rand);
    }
}

```

```

        sem_wait(&house_surplus);    //wait until have place to store
        sem_wait(&mutex);
        step_control++;
        if (step_control >= MAX_STEP)
            break;
        rand = (getrand(96) + 64);
        sem_post(&mutex);
        sem_post(&product_surplus); //produced, product--
    }
}

```

其中step_control为了控制演示演示步数，每个子进程最多演示MAX_STEP次就会终止，方便截图展示，MAX_STEP设为20。注意到的是，当缓冲区被存满后，就会变成任意生产者生产→消费者立即消费的单一循环，为了改变这种情况，将生产者的生产时间增大，从而使得缓冲区难以变满（供小于求）。

3.4.3. 哲学家就餐

修改位置：lab4/philosopher/main.c

这里有4个哲学家，因此，我们需要在在main函数里fork出4个子进程，执行哲学家函数。

本问题需要信号量数组forks[5]，表示每个位置的叉子，初始为1，表示有叉子。同时，为了避免死锁，需要引入mcount信号量，用于保证最多只有4个哲学家同时就餐，此时必然至少有一个可以就餐，从而避免了死锁。信号量的初始化如下所示：

```

for (int i = 0; i < 5; i++)
    sem_init(&forks[i], 1);
sem_init(&mcount, 4);

```

对于哲学家函数，其先**思考**，然后申请拿叉子（即对mcount进行wait操作），申请成功后每次尝试先后拿起左、右叉子，全部拿起则**用餐**，用餐结束后放下左右叉子，结束自己的申请。

```

void Philosopher(int id)
{
    while (1)
    {
        printf("Philosopher %d: think\n", id);
        sleep(getrand(128));
        sem_wait(&mcount);
        sem_wait(&forks[id - 1]); //left
        sem_wait(&forks[id % 5]); //right
        printf("Philosopher %d: eat\n", id);
        cnt++;
        if (cnt >= MAX)
            break;
        sleep(getrand(128));
        sem_post(&forks[id % 5]); //right
        sem_post(&forks[id - 1]); //left
        sem_post(&mcount);
    }
}

```

其中cnt和MAX的作用与3.4.2.中的step_control类似，用于控制最大执行步数，这里MAX设为5。

3.4.4. 读者-写者

修改位置：lab4/reader_writer/main.c

对于3个读者，3个写者，需要申请6个子进程，前3个为写者、后3个为读者。同时需要2个信号量 write_mutex，用来控制最多一个写者操作；Rcount_mutex，用于统计读者个数时独占。这两个信号量的初始化如下：

```
sem_init(&write_mutex, 1);
sem_init(&Rcount_mutex, 1);
```

按照设定，当某一个读者发现自己是第一个读者时，开始禁止写者进行写，同时开始读，然后其他的读者也可以入驻，当某个读者读结束后发现自己是最后一个读者，开始同意写者进行写。然而，这种方法是读者优先的，随着执行步数的增加，就会陷入到不停地有读者入驻，从而使得写者出现饥饿现象。

为了解决这个问题，我们可以引入新的信号量RW_mutex，它控制了同时在线的读者和写者的数量，而只要将其初值设为3，就可以解决上述饥饿问题，因为此时，当3个读者全部在读时，阻塞的下一个进程必然是写者的，从而确保了当他们读取完成后轮到某一个写者进行写操作，虽然这样总的来说依然是读者优先，但是对于这种读者和写者数量近似的情况来说，并不会导致写者长时间等待。

此外还需要注意的是，由于需要在读者读时输出在线的读者数量，还需要借用进程通信的相关内容，将读者数量这一参数的读/写放在共享内存中进行。这里使用到了共享内存的读写函数 read() 和 write()。

改进后的写者函数较为简单，如下所示：

```
void writer(int id)
{
    while (1)
    {
        sem_wait(&RW_mutex);
        sem_wait(&write_mutex);
        printf("writer %d: write\n", id);
        sleep(getrand(128));
        sem_post(&write_mutex);
        sem_post(&RW_mutex);
    }
}
```

读者函数相对较为复杂，它首先进行：

```
sem_wait(&RW_mutex);
sleep(getrand(64));
sem_post(&RW_mutex);
```

以确保写者不会饥饿，然后查看当前的在线读者数：

```

sem_wait(&Rcount_mutex);
read(SH_MEM, (uint8_t *)&Rcount, 4, 0);
Rcount++;
write(SH_MEM, (uint8_t *)&Rcount, 4, 0);
if (Rcount == 1)    //first
{
    sem_wait(&write_mutex);
}
sem_post(&Rcount_mutex);

```

当自己是第一个时，开始阻塞写者进程，并开始进行读操作：

```

printf("Reader %d: read, total %d reader\n", id, Rcount);
sleep(getrand(64));

```

然后，结束读操作，当自己是最后一个时，允许写进程进行操作：

```

sem_wait(&Rcount_mutex);
read(SH_MEM, (uint8_t *)&Rcount, 4, 0);
Rcount--;
write(SH_MEM, (uint8_t *)&Rcount, 4, 0);
if (Rcount == 0)    //last
{
    sem_post(&write_mutex);
}
sem_post(&Rcount_mutex);

```

这个实现避免了写者的饥饿，但是还有一个小问题，就是当最后一个读者离开时，很大概率所有的写者均被依次阻塞，从而导致步数足够大之后，写者的出现往往会按顺序依次进行而中间不会有读者参与，正如2.4.3中的截图展示的那样。