

---

# LAB4 实验报告

张运吉 (211300063、211300063@smail.nju.edu.cn)

(南京大学人工智能学院, 南京 210093)

## 1 实验进度

我已经完成 **LAB4** 全部必做内容和选做内容。

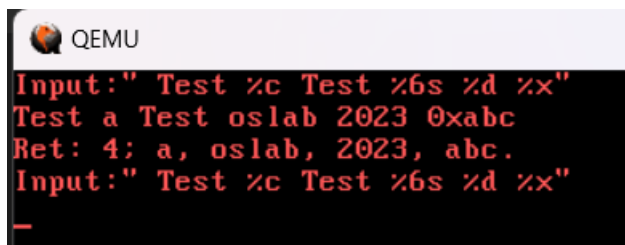
## 2 如何测试

我使用 5 个宏定义作为开关来决定是否测试某一部分。

```
// 一些用于控制测试的宏 main.c
#define TEST_SCANF // 测试scanf
#define TEST_SEM // 测试信号量
#define TEST_PHI_EATING // 测试哲学家进餐问题
#define TEST_PC_PROBLEM // 测试生产者-消费者问题
#define TEST_WR_PROBLEM // 测试读者-写着问题
```

## 3 实验结果

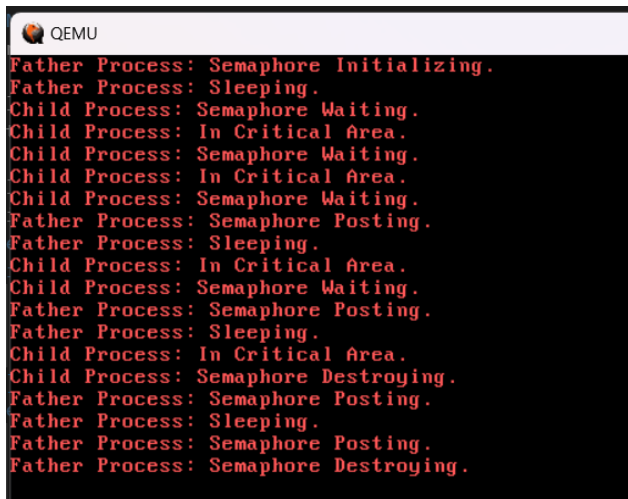
### 3.1 scanf



A screenshot of a QEMU terminal window. The title bar shows the QEMU logo and the text 'QEMU'. The terminal output is as follows:

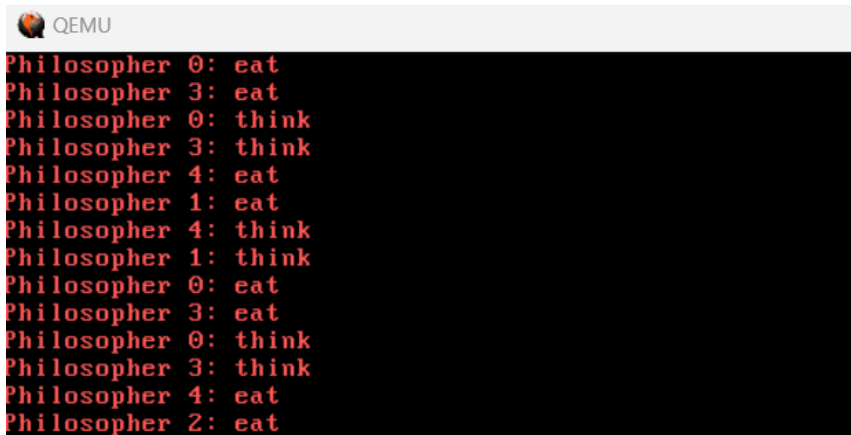
```
Input: "Test %c Test %6s %d %x"
Test a Test oslab 2023 0xabc
Ret: 4: a, oslab, 2023, abc.
Input: "Test %c Test %6s %d %x"
_
```

### 3.2 信号量



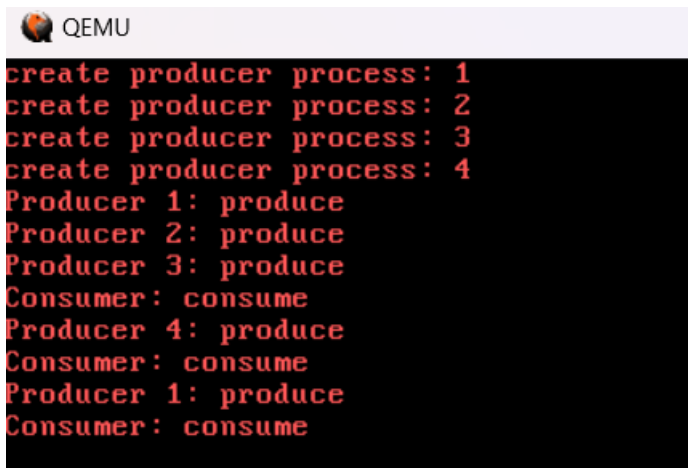
```
QEMU
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
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.
```

### 3.3 哲学家进餐



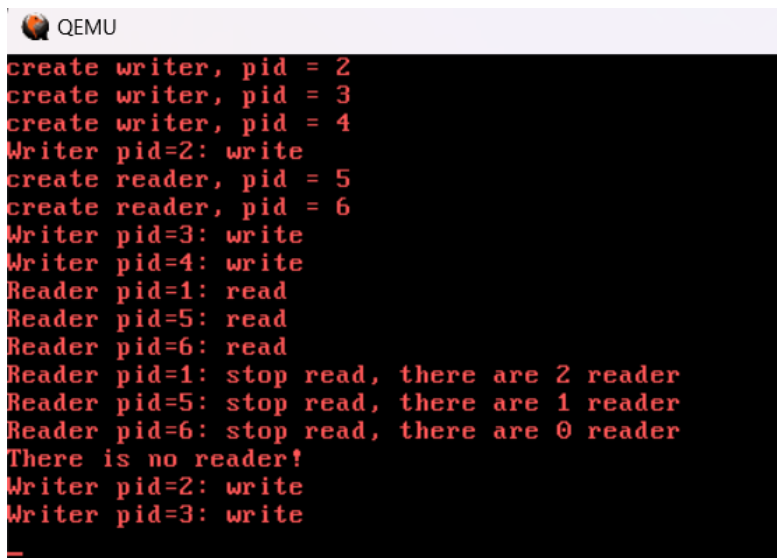
```
QEMU
Philosopher 0: eat
Philosopher 3: eat
Philosopher 0: think
Philosopher 3: think
Philosopher 4: eat
Philosopher 1: eat
Philosopher 4: think
Philosopher 1: think
Philosopher 0: eat
Philosopher 3: eat
Philosopher 0: think
Philosopher 3: think
Philosopher 4: eat
Philosopher 2: eat
```

### 3.4 生产者-消费者问题



```
QEMU
create producer process: 1
create producer process: 2
create producer process: 3
create producer process: 4
Producer 1: produce
Producer 2: produce
Producer 3: produce
Consumer: consume
Producer 4: produce
Consumer: consume
Producer 1: produce
Consumer: consume
```

### 3.5 读者-写者问题



```

QEMU
create writer, pid = 2
create writer, pid = 3
create writer, pid = 4
Writer pid=2: write
create reader, pid = 5
create reader, pid = 6
Writer pid=3: write
Writer pid=4: write
Reader pid=1: read
Reader pid=5: read
Reader pid=6: read
Reader pid=1: stop read, there are 2 reader
Reader pid=5: stop read, there are 1 reader
Reader pid=6: stop read, there are 0 reader
There is no reader!
Writer pid=2: write
Writer pid=3: write

```

## 4 TASK 关键代码实现

### 4.1 实现格式化输入函数

#### 4.1.1 keyboardHandle 函数

大致思路是：首先将读取到的 `keyCode` 放入到 `keyBuffer` 中，然后唤醒阻塞在 `dev[STD_IN]` 上的一个进程(`dev[STD_IN] < 0`)。

#### 4.1.2 syscallReadStdIn 函数

一个进程调用 `READ_STDIN` 的大致过程如下：尝试访问 `STDIN`，如果 `dev[STD_IN].value == 0`，阻塞该进程，然后模拟时钟中断进程切换，当检测到键盘中断时，唤醒该进程并读取 `keyBuffer` 中的所有数据；如果 `dev[STD_IN].value < 0`，返回-1。这里需要注意的是如果想要在终端看到输入的字符，那么需要在读取 `keyBuffer` 过程中通过 `stdout` 的方式输出字符。

### 4.2 实现信号量

#### 4.2.1 syscallSemInit 函数

这个函数是信号量初始化的处理例程，其功能就是在数组 `sem` 中查找一个未使用的信号量，根据参数初始化信号量的值 `value`，并设置好双向链表 `pcb`。在框架代码中已经给出。

#### 4.2.2 syscallSemWait 函数

这个函数是 P 操作对应的处理例程，如果尝试操作的信号量合法，先让信号量的值减一，如果信号量的值因此小于 0，那么把该进程放入信号量的阻塞队列，阻塞该进程，模拟时钟中断进行进程切换，返回 0；如果尝试操作的信号量不合法，那么返回-1。

#### 4.2.3 syscallSemPost 函数

这个函数是 V 操作对应的处理例程，同样要先对操作的信号量是否合法进行检查，在合法的情况下，使信号量的值加一，如果此时信号量的值 $\leq 0$ ，说明有进程阻塞在该信号量上，那么需要唤醒阻塞在该信号量上的一个进程(此部分框架代码已经给出)，操作成功后返

回 0；如果信号量不合法则返回-1.

#### 4.2.4 syscallSemDestory 函数

这个函数是销毁信号量的处理例程，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。具体销毁过程为直接将信号量的 state 清零。

### 4.3 哲学家就餐问题

在 main.c 中加了两个相关的函数：

```
void philosopher();
void test_philosopher();
```

第一个函数是哲学家进餐的实现，我这里使用讲义上的方案三，讲义上已经给出了伪代码，需要做的就是将伪代码写成相应的 c 代码，具体实现就是，首先定义一个信号量数组 forks，这些信号量都初始化为 1，每个信号量控制一把叉子，然后通过给出的思路调用之前实现的信号量 API 实现函数即可。

```
void philosopher() {
    int i = get_pid() - 1;
    while (1) {
        if (i % 2 == 0) {
            sem_wait(&forks[i]);
            sleep(128);
            sem_wait(&forks[(i + 1) % N]);
        }
        else {
            sem_wait(&forks[(i + 1) % N]);
            sleep(128);
            sem_wait(&forks[i]);
        }
        printf("Philosopher %d: eat\n", i);
        sleep(128); // eat
        printf("Philosopher %d: think\n", i);
        sem_post(&forks[i]);
        sleep(128);
        sem_post(&forks[(i + 1) % N]);
        sleep(128); // think
    }
}
```

第二个函数是用来测试的，这个函数首先通过 fork 创建 4 个子进程，然后这 4 个子进程和父进程并发运行，就达到了 5 个哲学家进程一起运行的效果。

#### 4.4 生产者-消费者问题(选做)

在 main.c 中有三个相关的函数：

```
void producer(sem_t *mutex, sem_t *empty, sem_t *full);
void consumer(sem_t *mutex, sem_t *empty, sem_t *full);
void producer_consumer();
```

此部分的伪代码讲义已经给出，具体实现就是初始化三个信号量：mutex,full,empty,其

中 `mutex` 初始化为 1，用来控制同一时刻只能有一个生产者或消费者进程在临界区，`full` 和 `empty` 作为条件变量，来控制生产者和消费者进程的执行顺序，`full` 初始化为 0，代表此时没有产品，在消费者进程进入临界区之前，先对这个信号量进行 P 操作，判断缓冲区是否存在可消费的产品，生产者进程在退出临界区之后进行 V 操作，表示产品增加；`empty` 初始化为 3，代表此时有三个可以存放产品的空位，生产者进程在进入临界区之前会调用 P 操作，检查是否有空缓冲区，消费者进程在退出临界区会调用 V 操作，在消费后增加空缓冲区的数量。

#### 4.5 读者-写者问题(选做)

在 `main.c` 中有三个相关的函数：

```
void writer(sem_t *writeMutex);
void reader(sem_t *rcount, sem_t *writemutex, sem_t *countmutex);
void reader_writer();
```

写者：只需要检查 `writeMutex`，如果没有被锁就可以进入临界区，否则阻塞等待。

读者：首先要对 `countMutex` 执行 P 操作，保证只有一个进程可以对 `Rcount` 更改，然后检查 `Rcount`，判断是否有其他进程在读，然后对 `writeMutex` 进行 P 操作，保证写互斥，然后使 `Rcount` 的值+1，释放 `countMutex` 让其他进程可以更改 `Rcount`，进行临界区，离开临界区之后 `Rcount` 信号量-1，检查是否为 0，如果为 0 则说明临界区内没有读者了，此时可写，释放 `writeMutex` 信号量。

这里比较特殊的是 `Rcount` 这个变量，讲义里把它设置成信号量，但我们在实现的时候需要对这个信号量进行加减操作，而目前提供的信号量相关的接口又不能很好实现这个要求，于是我增加了几个信号量相关的接口，分别是：

```
int sem_inc(sem_t *sem) { // 简单对信号量的value加一
    return syscall(SYS_SEM, SEM_INC, *sem, 0, 0, 0);
}

int sem_dec(sem_t *sem) { // 简单对信号量的value减一
    return syscall(SYS_SEM, SEM_DEC, *sem, 0, 0, 0);
}

int sem_read(sem_t *sem) { // 读取信号量的value
    return syscall(SYS_SEM, SEM_READ, *sem, 0, 0, 0);
}
```

## 5 收获与感想

通过实现操作系统的信号量及对应的系统调用，更加深刻了解信号量机制；基于信号量解决哲学家就餐问题、生产者-消费者问题、读者-写者问题，提高了编写进程互斥、同步代码的能力。