

OS Lab

Lab4

191220008 陈南瞳

924690736@qq.com

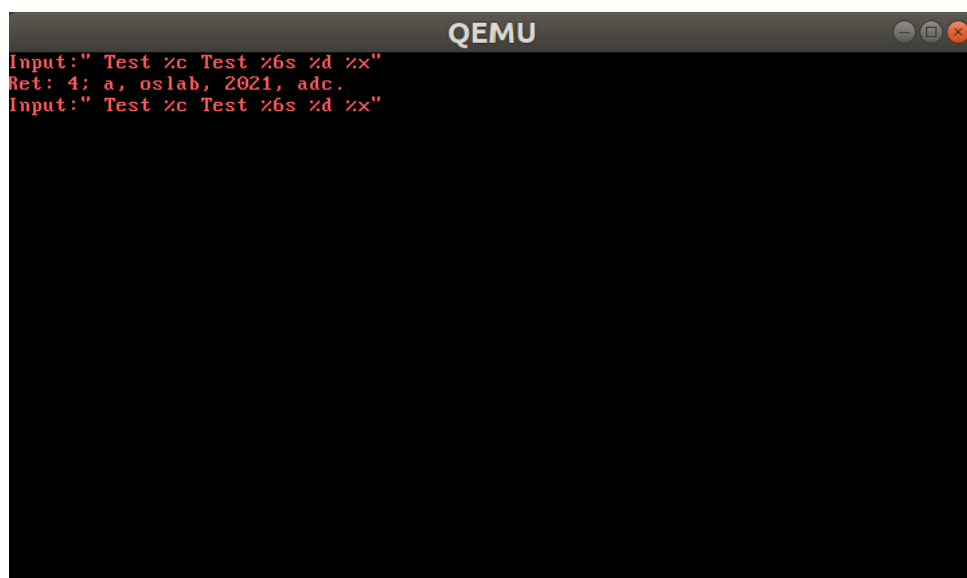
一、实验进度

完成了Lab4中的所有内容：

- 实现格式化输入函数
- 实现信号量相关系统调用
- 基于信号量解决进程同步问题：哲学家就餐问题，生产者-消费者问题，读者-写者问题

二、实验结果

(一) 格式化输入函数



```
QEMU
Input: " Test %c Test %6s %d %x"
Ret: 4: a, oslab, 2021, adc.
Input: " Test %c Test %6s %d %x"
```

(二) 信号量相关系统调用

```
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.
```

(三) 基于信号量解决进程同步问题

1、哲学家就餐问题

```
QEMU
philosopher
Philosopher 0: think
Philosopher 1: think
Philosopher 2: think
Philosopher 3: think
Philosopher 4: think
Philosopher 0: pick up left fork
Philosopher 1: pick up right fork
Philosopher 2: pick up left fork
Philosopher 3: pick up right fork
Philosopher 4: pick up left fork
Philosopher 0: pick up right fork
Philosopher 1: pick up left fork
Philosopher 3: pick up left fork
Philosopher 0: eat
Philosopher 3: eat
Philosopher 0: think
Philosopher 1: eat
Philosopher 3: think
Philosopher 4: pick up right fork
```

2、生产者-消费者问题

```
QEMU
producer_consumer
Producer 1: produce
Producer 2: produce
Producer 3: produce
Producer 4: produce
Consumer : consume
Producer 1: produce
Producer 2: produce
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
Producer 4: produce
Consumer : consume
```

3、读者-写者问题

```
QEMU
reader_writer
Writer 3: write
Writer 4: write
Writer 5: write
Reader 0: read, total 1 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
Writer 3: write
Writer 4: write
Writer 5: write
Reader 0: read, total 1 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
Writer 3: write
Writer 4: write
Writer 5: write
Reader 0: read, total 1 reader
Reader 1: read, total 2 reader
Reader 2: read, total 3 reader
```

三、实验修改的代码位置

(一) 格式化输入函数

由于 `syscall.c` 中的 `scanf` 已经完成，因此只需要完成对应的中断处理例程。

在 `kernel/irqHandle.c` 中，完成函数 `syscallReadStdIn` 和 `keyboardHandle` 的编写。

`keyboardHandle`：

- 将读取到的 `keyCode` 放入到 `keyBuffer` 中。
- 唤醒阻塞在 `dev[STD_IN]` 上的一个进程。

`syscallReadStdIn`：

- 如果 `dev[STD_IN].value == 0`，将当前进程阻塞在 `dev[STD_IN]` 上
- 进程被唤醒，读 `keyBuffer` 中的所有数据

最多只能有一个进程被阻塞在 `dev[STD_IN]` 上，多个进程想读，那么后来的进程会返回 -1，其他情况 `scanf` 的返回值应该是实际读取的字节数。

(二) 信号量相关系统调用

在 `kernel/irqHandle.c` 中，完成四个子例程的编写：`syscallSemInit`、`syscallSemwait`、`syscallSemPost` 和 `syscallSemDestroy`。

将 `current` 线程加到信号量 `i` 的阻塞列表可以通过以下代码实现

```
pcb[current].blocked.next = sem[i].pcb.next;
pcb[current].blocked.prev = &(sem[i].pcb);
sem[i].pcb.next = &(pcb[current].blocked);
(pcb[current].blocked.next)->prev = &(pcb[current].blocked);
```

以下代码可以从信号量 `i` 上阻塞的进程列表取出一个进程：

```
pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) - (uint32_t)&
(((ProcessTable*)0)->blocked));
sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
(sem[i].pcb.prev)->next = &(sem[i].pcb);
```

(三) 基于信号量解决进程同步问题

为了方便区分进程，我首先实现了 `getpid` 系统调用，用来返回当前进程的 `pid`。

在 `kernel/irqHandle.c` 中：

```
#define SYS_GETPID 7
...
void syscallHandle(struct StackFrame *sf) {
    switch(sf->eax) { // syscall number
        ...
        case SYS_GETPID:
            syscallGetPid(sf);
            break; // for SYS_GETPID
        default: break;
    }
}
...
void syscallGetPid(struct StackFrame *sf) {
    pcb[current].regs.eax = current;
    return;
}
```

在 `lib/syscall.c` 中:

```
pid_t getpid() {  
    return syscall(SYS_GETPID, 0, 0, 0, 0, 0);  
}
```

1、哲学家就餐问题

问题描述:

- 5个哲学家围绕一张圆桌而坐
 - 桌子上放着5支叉子
 - 每两个哲学家之间放一支
- 哲学家的动作包括思考和进餐
 - 进餐时需要同时拿到左右两边的叉子
 - 思考时将两支叉子返回原处
- 如何保证哲学家们的动作有序进行? 如: 不出现有人永远拿不到叉子

解决方案1有可能会死锁: 当五位哲学家都取到左边(右边)的叉子后, 下一位哲学家取右边的叉子后, 会出现无限等待的情况, 发生死锁。

解决方案2通过使用信号量 `mutex` 的方法, 解决了死锁的问题, 但同时仅允许一位哲学家进餐。

解决方案3通过让编号为偶数的哲学家只能先取左边的叉子再取右边的叉子, 编号为奇数的哲学家只能先取右边的叉子再取左边的叉子, 成功解决方案1和方案2中的问题。

在 `app/main.c` 中编写函数 `philosopher`。

2、生产者-消费者问题

问题描述:

- 一个或多个生产者在生产数据后放在一个缓冲区里
- 单个消费者从缓冲区取出数据处理
- 任何时刻只能有一个生产者或消费者可访问缓冲区

问题分析:

- 任何时刻只能有一个线程操作缓冲区(互斥访问)
- 缓冲区空时, 消费者必须等待生产者(条件同步)
- 缓冲区满时, 生产者必须等待消费者(条件同步)

用信号量描述每个约束:

- 二进制信号量 `mutex`: 控制对产品临界区的访问
- 资源信号量 `full`: 控制对产品临界区的访问
- 资源信号量 `empty`: 控制对产品临界区的访问

在 `app/main.c` 中编写函数 `producer_consumer`。

3、读者-写者问题

问题描述：

- 共享数据的两类使用者
 - 读者：只读取数据，不修改
 - 写者：读取和修改数据
- 对共享数据的读写
 - “读-读”允许，同一时刻，允许有多个读者同时读
 - “读-写”互斥，没有写者时读者才能读，没有读者时写者才能写
 - “写-写”互斥，没有其他写者，写者才能写

用信号量描述每个约束：

- 信号量 `writelock`，控制读写操作的互斥，初始化为1
- 读者计数 `readcount`，正在进行读操作的读者数目，初始化为0
- 信号量 `mutex`，控制对读者计数的互斥修改，初始化为1，只允许一个线程修改 `readcount` 计数

在 `app/main.c` 中编写函数 `reader_writer`。

四、遇到的问题和对这些问题的思考

1、在实现了哲学家就餐问题，生产者-消费者问题和读者-写者问题后，均产生了难以解释的死锁现象。

经检查，代码逻辑并没有问题，通过 debug，发现是信号量的初始化出现了问题。

最初，对于信号量的初始化的方式是：

```
sem_t *x = {0};  
...  
sem_init(x, 0 or 1);
```

后来发现这种方式并没有对 `x` 进行成功的初始化，因为 `x` 并未被分配确切的内存地址，对其进行赋值也就没有意义。所以改用下面这种方式：

```
sem_t _x = 0;  
sem_t *x = &_x;  
...  
sem_init(x, 0 or 1);
```

通过定义 `_x`，赋以初值0，再将 `x` 的指向 `_x`，因此 `x` 被分配了对应的内存地址，便能进行正确的初始化。

2、通过问题1中的处理后，哲学家就餐问题和生产者-消费者问题都成功地消除了死锁现象，但读者-写者问题仍然会死锁，无法正常运行。

在排除代码本身逻辑错误的可能性后，我将出现问题的原因聚焦在了读者-写者问题与其它两个问题不同的地方。经比较，发现读者-写者问题中用到了其它两个问题都没有用到的全局变量 `readcount`。经过简单的检验之后，发现的确是 `readcount` 的更新出现错误导致了死锁。起初我也无法解释这种现象，请教了其他同学后，才知道是框架代码存在的问题：在 `enTry` 函数外面设置的变量并不能被实现成为全局变量。所以我们必须在内核 `syscall.c` 文件中设置出一段空间用于各个进程的全局变量的内容的共享。

我在 `syscall.c` 中设置了一个内核共享的变量 `count` 用于实现各个进程的 `readcount` 的值的共享，即在对 `readcount` 值进行修改的时候就先从内核中把值读取出来，然后修改完成后再写回去，这样就能实现进程的共享变量的实现。同时因为这里的 `readcount` 在 `main.c` 中已经为其加了锁，所以内核代码中并不会造成同时访问内核中 `count` 的情况。

```
readcount = read();
readcount change;
write(readcount);
```

3、部分进程始终无法出现。

信号量数量最大值过小，因此在 `meomory.h` 中修改：

```
// #define MAX_SEM_NUM 8
#define MAX_SEM_NUM 10
```

五、对讲义或框架代码中某些思考题的看法

思考题

1、有没有更好的方式处理这个就餐问题？

方案一：至多允许4位哲学家同时进餐。

方案二：每位哲学家取到手边的两把叉子才开始进餐，否则一把叉子也不取。

2、P、V的操作顺序有影响吗？

- P (full) / P (empty) 与 P (mutex) 的顺序不可颠倒，必须先对资源信号量进行 P 操作，再对互斥信号量进行 P 操作，否则会导致死锁。（若此时缓冲区已满，而生产者先 P (mutex), 取得缓冲池访问权，再 P (empty), 由于缓冲池已满，empty = 0, 导致 P (empty) 失败，生产者进程无法继续推进，始终掌握缓冲池访问权无法释放，因而消费者进程无法取出产品，导致死锁）
- V (full) / V (empty) 与 V (mutex) 的顺序可以颠倒。

六、实验心得或对提供帮助的同学的感谢

经过徐峰老师上次的编程作业和期中考试复习后，对于信号量和PV操作较为熟悉，因此本次实验整体不难理解。再加上教程中有一些伪代码的提示，实验中的效率也更高，希望以后也能有类似的提示。

但本次实验仍然遇到了不少的问题，通过自己 debug 和请教其他同学的方式，最终得以解决。