

最后修改: 2020-4-26

1. 实验要求

本实验要求实现操作系统的信号量及对应的系统调用，然后基于信号量解决生产者-消费者、哲学家就餐、读者-写者问题

1.1. 实现格式化输入函数

在lab2中就要求大家实现了格式化输出函数，但是格式化输入函数一直不见踪影，为什么呢？原因之一是基于中断的 `scanf` 需要进行进程同步，而这个在前面的实验中没有涉及，本实验首先需要大家实现一个 `scanf` 格式化输入函数并使用以下代码进行测试，为进程同步内容打下基础：

```
#include "lib.h"
#include "types.h"

int uEntry(void) {
    int dec = 0;
    int hex = 0;
    char str[6];
    char cha = 0;
    int ret = 0;
    while(1){
        printf("Input:\n Test %%c Test %%6s %%d %%x\\n\\n");
        ret = scanf(" Test %c Test %6s %d %x", &cha, str, &dec, &hex);
        printf("Ret: %d; %c, %s, %d, %x.\\n", ret, cha, str, dec, hex);
        if (ret == 4)
            break;
    }
    return 0;
}
```

输入 `Test a Test oslab 2020 0xadc` 后，屏幕上输出为 `Ret: 4; a, oslab, 2020, adc.`

要求非格式字符原样输入，其它异常（输入和要求格式不符等）可自行斟酌决定如何处理

1.2. 实现进程通信

借鉴管道和共享内存的概念，基于 `read` 和 `write` 实现一个简单的进程通信机制，使用如下测试用例进行测试

```
#include "lib.h"
#include "types.h"

int uEntry(void) {
```

```

int data = 2020;
int data1 = 1000;
int i = 4;
int ret = fork();
if (ret == 0) {
    while (i != 0) {
        i--;
        printf("Child Process: %d, %d\n", data, data1);
        write(SH_MEM, (uint8_t *)&data, 4, 0);          // define SH_MEM 3
        data += data1;
        sleep(128);
    }
    exit();
} else if (ret != -1) {
    while (i != 0) {
        i--;
        read(SH_MEM, (uint8_t *)&data1, 4, 0);
        printf("Father Process: %d, %d\n", data, data1);
        sleep(128);
    }
    exit();
}
return 0;
}

```

一个可能的输出：

```

QEMU
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
-

```

1.3. 实现信号量相关系统调用

实现 `SEM_INIT`、`SEM_POST`、`SEM_WAIT`、`SEM_DESTROY` 系统调用，使用以下用户程序测试，并在实验报告中说明实验结果

```

#include "lib.h"
#include "types.h"

int uEntry(void) {
    int i = 4;
    int ret = 0;
    int value = 2;

    sem_t sem;
    printf("Father Process: Semaphore Initializing.\n");
    ret = sem_init(&sem, value);
    if (ret == -1) {
        printf("Father Process: Semaphore Initializing Failed.\n");
        exit();
    }

    ret = fork();
    if (ret == 0) {
        while( i != 0) {
            i--;
            printf("Child Process: Semaphore Waiting.\n");
            sem_wait(&sem);
            printf("Child Process: In Critical Area.\n");
        }
        printf("Child Process: Semaphore Destroying.\n");
        sem_destroy(&sem);
        exit();
    }
    else if (ret != -1) {
        while( i != 0) {
            i--;
            printf("Father Process: Sleeping.\n");
            sleep(128);
            printf("Father Process: Semaphore Posting.\n");
            sem_post(&sem);
        }
        printf("Father Process: Semaphore Destroying.\n");
        sem_destroy(&sem);
        exit();
    }

    return 0;
}

```

一个可能的输出：

```
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.
```

1.4. 基于信号量解决进程同步问题

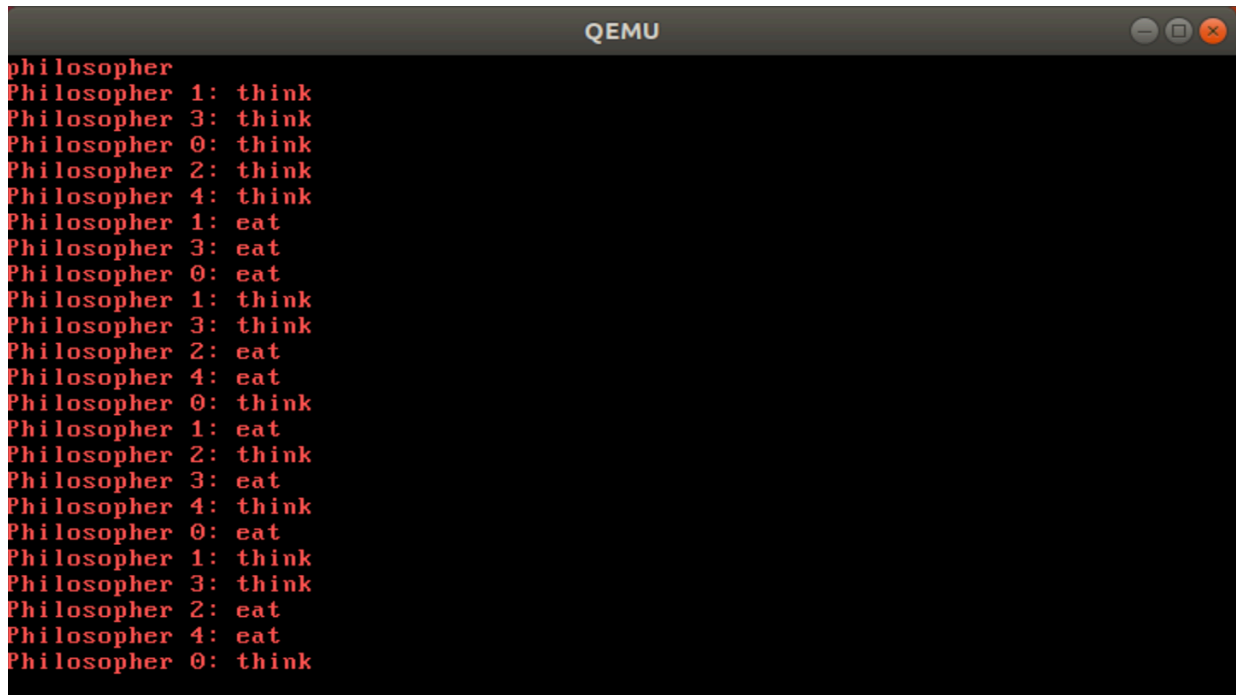
基于信号量解决生产者-消费者、哲学家就餐、读者-写者问题，将每个问题实现成一个用户程序并写入文件系统，用户可以通过键盘输入选择想执行的用户程序，最终提交代码的 `lab4/app/main.c` 中的代码如下：

```
#include "lib.h"
#include "types.h"

int uEntry(void) {
    char ch;
    printf("Input: 1 for bounded_buffer\n          2 for philosopher\n          3 for reader_writer\n");
    scanf("%c", &ch);
    switch (ch) {
        case '1':
            exec("/usr/bounded_buffer", 0);
            break;
        case '2':
            exec("/usr/philosopher", 0);
            break;
        case '3':
            exec("/usr/reader_writer", 0);
            break;
        default:
            break;
    }
    exit();
    return 0;
}
```

```
}
```

哲学家就餐问题的一个可能输出如下：



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

2. 相关资料

2.1. 进程通信

进程通信可划分为阻塞或非阻塞，共享内存就是非阻塞进程通信的一种，共享内存是把同一个物理内存区域同时映射到多个进程的内存地址空间的通信机制，它只在通信前显式的设置共享内存段，通信时不需要通过系统调用切换内核态，可以快速方便地共享数据；但是在linux系统的实现中，共享内存需要依赖于页表，在我们的实验中，采用分段内存管理，很难直接进行共享内存

考虑到进程通信对操作系统的重要性，我们结合共享内存和管道的机制，将共享内存直接放到内核中，借用文件读写接口直接对内核中的共享内存进行操作

目前的操作系统还没有文件读写功能，我们需要添加两个系统调用用于读写文件

```
int read(int fd, uint8_t *buffer, int size);
int write(int fd, uint8_t *buffer, int size);
```

其中，fd是文件标识符，buffer为读和写的内容，size是想读或写的内容的大小

考虑到直接使用读写接口模拟共享内存，在调用read和write时，还要传入一个index指示我想读写的共享内存的位置，也就是说，目前需要实现的系统调用如下

```
int read(int fd, uint8_t *buffer, int size, int index);
int write(int fd, uint8_t *buffer, int size, int index);
```

两个系统调用的返回值表示的是实际读写的字节数

2.2. 信号量

相信课上大家已经对信号量有了一定的了解，这里以另一个角度介绍一下信号量

信号是一种抽象数据类型，由一个整型（sem）变量和两个原子操作组成：

`P()`（Prolaag，荷兰语尝试减少）

- sem减1
- 如sem<0，进入等待，否则继续

`V()`（Verhoog，荷兰语增加）

- sem加1
- 如sem<=0，唤醒一个等待进程

信号量的实现（伪代码）：

```
class Semaphore {
    int sem;
    WaitQueue q;
}

Semaphore::P(){
    sem--;
    if(sem < 0){
        Add this thread t to q;
        block(t)
    }
}

Semaphore::V(){
    sem++;
    if(sem <= 0){
        Remove a thread t from q;
        wakeup(t);
    }
}
```

2.2.1. 用信号量实现临界区的互斥访问

每类资源设置一个信号量，其初值为1

```
mutex = new Semaphore(1);
mutex->P();
Critical Section;
mutex->V();
```

这里要注意必须成对使用 `P()` 操作和 `V()` 操作

- `P()` 操作保证互斥访问临界资源
- `V()` 操作在使用后释放临界资源
- `PV` 操作不能次序错误、重复或遗漏

2.2.2. 用信号量实现条件同步

条件同步设置一个信号量，其初值为0

```
condition = new Semaphore(0);
```

thread A	thread B
... M ...	
... N X ...
	... Y ...

A 有 M 和 N 模块，B 有 X 和 Y 模块，这里为了保证 B 执行到 X 后，A 才能执行 N，可以使用信号量实现条件同步

thread A		thread B
... M ...		
condition->P();	-----+	
... N X ...
	+----->	condition->V();
		... Y ...

2.2.3. 生产者-消费者问题

生产者---->缓冲区---->消费者

有界缓冲区的生产者-消费者问题描述：

- 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
- 资源信号量fullBuffers

- 资源信号量emptyBuffers

伪代码描述一下：

```
class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
```

```
BoundedBuffer::Deposit(c){
    emptyBuffers->P();
    mutex->P();
    Add c to the buffer;
    mutex->V();
    fullBuffers->V();
}
```

```
BoundedBuffer::Remove(c){
    fullBuffers->P();
    mutex->P();
    Remove c from buffer;
    mutex->V();
    emptyBuffers->V();
}
```

P、V的操作顺序有影响吗？

2.2.4. 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

方案1：

```
#define N 5 // 哲学家个数
semaphore fork[5]; // 信号量初值为1
void philosopher(int i){ // 哲学家编号：0-4
    while(TRUE){
        think(); // 哲学家在思考
        P(fork[i]); // 去拿左边的叉子
        P(fork[(i+1)%N]); // 去拿右边的叉子
        eat(); // 吃面条
        V(fork[i]); // 放下左边的叉子
        V(fork[(i+1)%N]); // 放下右边的叉子
    }
}
```

极端情况下不正确，可能导致死锁

方案2:

```
#define N 5                // 哲学家个数
semaphore fork[5];        // 信号量初值为1
semaphore mutex;          // 互斥信号量, 初值1
void philosopher(int i){  // 哲学家编号: 0-4
    while(TRUE){
        think();          // 哲学家在思考
        P(mutex);         // 进入临界区
        P(fork[i]);        // 去拿左边的叉子
        P(fork[(i+1)%N]);  // 去拿右边的叉子
        eat();             // 吃面条
        V(fork[i]);        // 放下左边的叉子
        V(fork[(i+1)%N]);  // 放下右边的叉子
        V(mutex);         // 退出临界区
    }
}
```

互斥访问正确, 但是每次只允许一个人就餐

方案3:

```
#define N 5                // 哲学家个数
semaphore fork[5];        // 信号量初值为1
void philosopher(int i){  // 哲学家编号: 0-4
    while(TRUE){
        think();          // 哲学家在思考
        if(i%2==0){
            P(fork[i]);    // 去拿左边的叉子
            P(fork[(i+1)%N]); // 去拿右边的叉子
        } else {
            P(fork[(i+1)%N]); // 去拿右边的叉子
            P(fork[i]);    // 去拿左边的叉子
        }
        eat();             // 吃面条
        V(fork[i]);        // 放下左边的叉子
        V(fork[(i+1)%N]);  // 放下右边的叉子
    }
}
```

没有死锁, 可以实现多人同时就餐

有没有更好的方式处理这个就餐问题?

2.2.5. 读者-写者问题

读者-写者问题主要出现在数据库等共享资源的访问当中, 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WriteMutex，控制读写操作的互斥，初始化为1
- 读者计数Rcount，正在进行读操作的读者数目，初始化为0
- 信号量CountMutex，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Rcount计数

写者进程

```
P(WriteMutex);
write;
V(WriteMutex);
```

读者进程

```
P(CountMutex);
if (Rcount == 0)
    P(WriteMutex);
++Rcount;
V(CountMutex);
read;
P(CountMutex);
--Rcount;
if (Rcount == 0)
    V(WriteMutex);
V(CountMutex);
```

2.2.6. 相关系统调用

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);
```

3. 实验攻略

在攻略之前，先带大家看一看实验4新增的或修改的数据结构等：

新增数据结构

```
struct Semaphore {
    int state;
    int value;
    struct ListHead pcb; // link to all pcb ListHead blocked on this semaphore
};
typedef struct Semaphore Semaphore;

struct Device {
    int state;
    int value;
    struct ListHead pcb; // link to all pcb ListHead blocked on this device
};
typedef struct Device Device;

Semaphore sem[MAX_SEM_NUM];
Device dev[MAX_DEV_NUM];
```

信号量 `Semaphore` 相信看完[相关资料](#)，大家都能理解，因为我们将信号量的定义成数组，所以添加了一个 `state` 成员，表示当前信号量是不是正在使用，1表示正在使用，0表示未使用

那么 `Device` 是干啥用的，为什么和信号量的定义这么相似，说这个之前我们需要想一下 `stdin` 标准输入，在操作系统中，我们不可能通过一直监听键盘中断来进行输入，这样太浪费系统资源了，所以我们需要一个键盘输入缓冲区和类似信号量的东西来实现条件同步，在键盘中断将输入存入缓冲区后再让用户程序读取，所以代码中定义了 `Device`，他其实就是信号量，只不过不能由用户通过系统调用控制，而是直接和硬件绑定

在实验中，我们将stdin, stdout, sharedmemory都抽象成了Device，其中

```
#define STD_OUT 0
#define STD_IN 1
#define SH_MEM 3
```

实际上，stdout和sharedmemory都是非阻塞式的，也就是说只有stdin上才会有进程阻塞，而stdout和sharedmemory都是非阻塞式

```
struct ListHead {
    struct ListHead *next;
    struct ListHead *prev;
};
```

ListHead 是一个双向链表

如下两个函数用于初始化 sem 和 dev

```
void initSem() {
    int i;
    for (i = 0; i < MAX_SEM_NUM; i++) {
        sem[i].state = 0; // 0: not in use; 1: in use;
        sem[i].value = 0; // >=0: no process blocked; -1: 1 process blocked; -2: 2
process blocked;...
        sem[i].pcb.next = &(sem[i].pcb);
        sem[i].pcb.prev = &(sem[i].pcb);
    }
}

void initDev() {
    int i;
    for (i = 0; i < MAX_DEV_NUM; i++) {
        dev[i].state = 1; // 0: not in use; 1: in use;
        dev[i].value = 0; // >=0: no blocked; -1: 1 process blocked; -2: 2 process
blocked;...
        dev[i].pcb.next = &(dev[i].pcb);
        dev[i].pcb.prev = &(dev[i].pcb);
    }
}
```

修改的数据结构：PCB中添加对应的双向链表结构

```

struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct TrapFrame regs;
    uint32_t stackTop;
    uint32_t prevStackTop;
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    char name[32];
+ struct ListHead blocked; // semaphore, device, file blocked on
};
typedef struct ProcessTable ProcessTable;

```

这样将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);

```

irqHandle.c中的syscallWrite也有一些变化：

```

void syscallWrite(struct TrapFrame *tf) {
    switch(tf->ecx) { // file descriptor
-   case 0:
-       syscallPrint(tf);
+   case STD_OUT:
+       if (dev[STD_OUT].state == 1) {
+           syscallWriteStdOut(tf);
+       }
+       break; // for STD_OUT
+   case SH_MEM:
+       if (dev[SH_MEM].state == 1) {
+           syscallWriteShMem(tf);
+       }
+       break; // for SH_MEM
    default: break;
    }
}

```

一是对syscallPrint重命名

```
-void syscallPrint(struct TrapFrame *tf) {  
+void syscallWriteStdOut(struct TrapFrame *tf) {
```

然后就是添加了待实现的 `syscallWriteShMem`，用于写内核共享内存

3.1. 实现格式化输入函数

为了降低实验难度，`syscall1.c` 中的 `scanf` 已经完成，同学们只需要完成对应的中断处理例程

`irqHandle.c`中添加了`syscallRead`函数处理各种读数据：

```
void syscallRead(struct TrapFrame *tf) {  
    switch(tf->ecx) {  
        case STD_IN:  
            if (dev[STD_IN].state == 1) {  
                syscallReadStdIn(tf);  
            }  
            break;  
        case SH_MEM:  
            if (dev[SH_MEM].state == 1) {  
                syscallReadShMem(tf);  
            }  
            break;  
        default:  
            break;  
    }  
}
```

在这一节主要关注的就是 `syscallReadStdIn`，同学们需要去完成它，那么如何完成呢，它是和键盘中断有条件同步的，所以的这一步还要结合 `keyboardHandle` 一起完成

在实验2中，有很多同学不知道下面的代码是干啥的

```
extern uint32_t keyBuffer[MAX_KEYBUFFER_SIZE];  
extern int bufferHead;  
extern int bufferTail;
```

其实这就是键盘输入的缓冲区，把所有零碎的知识拼凑在一起，`keyboardHandle` 要做的事情就两件：

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

接下来安排 `syscallReadStdIn`，它要做的事情也就两件：

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

值得注意的就是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

和实验2中printf的处理例程类似，以下代码可以将读取的字符 `character` 传到用户进程

```
int sel = tf->ds;
char *str = (char *)tf->edx;
int i = 0;
asm volatile("movw %0, %%es"::"m"(sel));
asm volatile("movb %0, %%es:(%1)"::"r"(character), "r"(str + i));
```

完成这一步后请测试 `scanf`，并在实验报告展示结果

3.2. 实现进程通信

同样的，这一部分我们将 `syscall.c` 中的封装部分完成了，同学们仅仅需要完善处理例程，也就是 `syscallWriteShMem` 和 `syscallReadShMem`，由于这一部分不需要阻塞，所以相对来说十分容易实现。

我们在内核 `irqHandle.c` 中定义了一段大小为4K的内存做为共享内存：

```
uint8_t shMem[MAX_SHMEM_SIZE];
```

完成后请进行测试，并在实验报告展示结果

3.3. 实现信号量

这一部分也只需要完善处理例程，其它部分已经实现，所有的信号量相关调用有一个总的处理：

```
void syscallSem(struct TrapFrame *tf) {
    switch(tf->ecx) {
        case SEM_INIT:
            syscallSemInit(tf);
            break;
        case SEM_WAIT:
            syscallSemWait(tf);
            break;
        case SEM_POST:
            syscallSemPost(tf);
            break;
        case SEM_DESTROY:
            syscallSemDestroy(tf);
            break;
        default:break;
    }
}
```

需要完成的是4个子例程：`syscallSemInit`、`syscallSemWait`、`syscallSemPost` 和 `syscallSemDestroy`

在实现时，因为信号量以数组形式存在，所以只要一个下标就可以定位信号量

完成后请进行测试，并在实验报告中展示结果

3.4. 解决进程同步问题

为了方便区分进程，我们实现了 `getpid` 系统调用，该函数返回的是当前进程的 `pid`

3.4.1. 生产者-消费者问题

同学们需要在 `lab4/bounded_buffer/main.c` 中实现生产者-消费者问题

要求：

- 4个生产者，1个消费者同时运行
- 生产者生产，`printf("Producer %d: produce\n", id);`
- 消费者消费，`printf("Consumer : consume\n");`
- 任意P、V及生产、消费动作之间添加 `sleep(128);`

3.4.2. 哲学家就餐问题

同学们需要在 `lab4/philosopher/main.c` 中实现哲学家就餐问题

要求：

- 5个哲学家同时运行
- 哲学家思考，`printf("Philosopher %d: think\n", id);`
- 哲学家就餐，`printf("Philosopher %d: eat\n", id);`
- 任意P、V及思考、就餐动作之间添加 `sleep(128);`

3.4.3. 读者-写者问题

同学们需要在 `lab4/reader-writer/main.c` 中实现读者-写者问题

要求：

- 3个读者，3个写者同时运行
- 读者读数据，`printf("Reader %d: read, total %d reader\n", id, Rcount);`
- 写者写数据，`printf("Writer %d: write\n", id);`
- 任意P、V及读、写动作之间添加 `sleep(128);`

3.4.4. 随机函数（选做）

最后加上一个简单的附加题，前面的各进程中，都是 `sleep` 一段相同的时间，实现一个随机函数用于生成随机数，这样各处就能随机 `sleep`

3.5. 一点建议

- 一定要使用git进行版本管理
- 勇于试错，善用各种调试方法
- 不要盲目相信框架代码，框架代码只在有限范围内正确

4. 作业提交

- 本次作业需提交可通过编译的实验相关源码与报告,提交前请确认 `make clean` 过.
- 请大家在提交的实验报告中注明你的邮箱, 方便我们及时给你一些反馈信息.
- **学术诚信:** 如果你确实无法完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分.
- 请你在实验截止前务必确认你提交的内容符合要求(压缩包命名, 格式, 相关内容等), 你可以下载你提交的内容进行确认. 如果由于你的原因给我们造成了不必要的麻烦, 视情况而定, 在本次实验中你将会被扣除一定的分数, 最高可达50%.
- 实验不接受迟交, 一旦迟交按**学术诚信**给分.
- 其他问题参看 `index.pdf` 中的**作业规范与提交**一章
- 本实验最终解释权归助教所有

截止时间: 2020-6-1 23:55