

《操作系统实验》实验报告

实验名称：进程同步

姓名：胡育玮

学号：171860574

邮箱：yeevee@qq.com

班级：17 级计算机科学与技术系 2 班

一、实验内容

本实验通过实现一个简单的生产者消费者程序，介绍基于信号量的进程同步机制。

1. 内核: 提供基于信号量的进程同步机制，并提供系统调用 `sem_init` 、 `sem_post` 、 `sem_wait` 、 `sem_destroy`
2. 库: 对上述系统调用进行封装
3. 用户: 对上述库函数进行测试

二、实验过程

(1) 实现 SEM_INIT 、 SEM_POST 、 SEM_WAIT 、 SEM_DESTROY、 GET_PID 系统调用

检查 syscall.c 可知，框架代码已封装好这些系统调用的接口，我们只需在 irqhandle.c 中给出这些系统调用的具体实现即可。

sem_init(): 框架代码已实现好

sem_wait():

sem_wait()对应 P 操作，应将相应信号量-1，并根据减一后信号量的值决定是否挂起该进程。

在其具体的实现 syscallSemWait()中，先通过参数 sf 中的 edx 域来获取信号量的编号（由 syscall.c 中的 syscall()函数知信号量编号存入了栈帧 sf 的 edx 域中）。然后将对应信号量的值减一，若减一后小于 0 则把当前进程加入阻塞队列。阻塞队列由链表实现；设信号量编号为 tar，则链表头节点就是 sem[tar].pcb，后续的链表节点是 pcb 数组中的被阻塞的进程对应的数组项的 block 域：

```

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

```

图 1：信号量结构体中的 **pcb** 域为阻塞在该信号量上的进程链表的头节点。pcb 内有两个域，分别是指针变量 prev 和指针变量 next，分别代表链表中的上一节点和下一节点。

```

struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct StackFrame 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
    //struct ListHead children;
    //struct ListHead sibling;
};

```

图 2：进程结构体中的 blocked 域用来作为阻塞链表中的节点，表示相应的进程被阻塞。blocked 域内部一样包含 next 和 prev 指针，指向上一/下一节点。

把当前进程加入阻塞队列的过程：首先来到当前阻塞队列的末尾节点；然后把当前要阻塞的进程设为链表的新的末尾；最后设置该进程的 blocked 域的 next 和 prev 字段。

接着，设置当前进程的 state 域为 blocked，正式阻塞当

前进程；然后呼唤 `int $0x20` 系统调用，调出时钟中断，以调度别的进程来执行。**呼唤 `int $0x20` 来实现调度是模仿框架代码中的 `syscallSleep()` 的语句的。**

实现代码：

```
void syscallSemWait(struct StackFrame *sf)
{
    int tar = (int)sf->edx; //获取信号量的编号.由sem_init()知这个编号就是在sem数组中的下标
    sem[tar].value--;
    if (sem[tar].value < 0) //把当前进程加入阻塞队列
    {
        struct ListHead *temp = &sem[tar].pcb;
        while (temp->next != &sem[tar].pcb)
            temp = temp->next; //来到阻塞队列的末尾

        temp->next = &pcb[current].blocked; //将当前进程添加到阻塞队列的末尾
        pcb[current].blocked.prev = temp; //设置末尾进程的block变量的prev和next.注意，队列末尾进程的
        next域的值应设为信号量变量sem[tar]的pcb域的地址
        pcb[current].blocked.next = &sem[tar].pcb;

        pcb[current].state = STATE_BLOCKED; //阻塞当前进程

        asm volatile("int $0x20"); //模仿上面的syscallSleep(),通过调用时钟中断来切换别的进程
    }

    return;
}
```

`sem_post()`: `sem_post()` 对应 V 操作，应将相应信号量+1，并根据+1 后信号量的值决定后续动作。

具体的实现 `syscallSemPost()`：与 `syscallSemWait()` 类似，在响应信号量+1 后若信号量值 ≤ 0 则需要唤醒一个进程。在我的算法里总是唤醒阻塞链表里的首个进程。首先在进程表里找到这个要唤醒的进程；然后修改阻塞链表来剔除这个进程；最后设置其 `state` 域为 `runnable`，正式唤醒之。**注意此时**

不需要呼唤时钟中断来切换进程，因为当前进程依然是 **running** 状态。

实现代码：

```
void syscallSemPost(struct StackFrame *sf)
{
    int tar = (int)sf->edx;
    sem[tar].value++;
    if (sem[tar].value <= 0) //释放阻塞队列中的一个进程
    {
        struct ListHead *temp = &sem[tar].pcb;

        int i;
        for (i = 0; i < MAX_PCB_NUM; i++)
        {
            if (&pcb[i].blocked == temp->next) //找到要解除阻塞的进程：在我的算法里
                                                //该进程为阻塞链表里的第一个进程
            {
                break;
            }

            struct ListHead* mid = temp->next; //修改阻塞链表
            temp->next = mid->next;
            temp->next->prev = temp;

            pcb[i].state = STATE_RUNNABLE; //唤醒相应进程
        }
        return;
    }
}
```

sem_destroy(): 设置信号量状态为不可用（0）即可：

```
void syscallSemDestroy(struct StackFrame *sf)
{
    int tar = (int)sf->edx;
    assert(sem[tar].state == 1);
    sem[tar].state = 0;

    return;
}
```

getpid(): 该系统调用获取当前进程的进程号。

实现流程：首先在 syscall.c 中实现一个接口：

```
int getpid()
{
    return syscall(7, SEM_WAIT, 0, 0, 0, 0);
}
```

我为 getpid()调用分配的系统调用号是7。**在我的实现中，只需要系统调用号即可获取进程编号，因此调用 syscall()函数时的后面的参数无关紧要。**

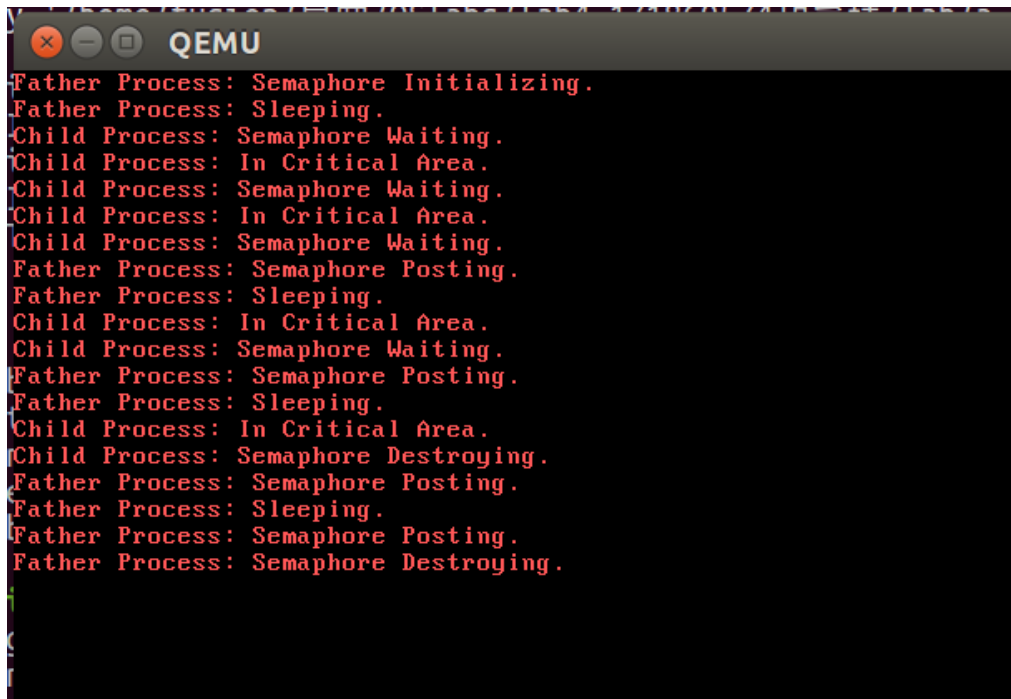
然后在 irqhandle.c 中的 syscallHandle ()中，根据系统调用号7来进行相应的操作获得进程号作为返回值：

```
case SYS_EXIT:
    syscallExit(sf);
    break; // for SYS_EXIT
case SYS_SEM:
    syscallSem(sf);
    break; // for SYS_SEM
case 7:
    pcb[current].regs.eax = pcb[current].pid;
    //asm volatile("movl %0, %%eax" ::"m"(current));
    break;
default:
    break;
}
```

将当前进程号赋给 PCB 中保存的现场的 eax 字段，这样到时候恢复现场后 eax 寄存器里就保存了当前进程号，于是就可以作为 syscall()的返回值传给 getpid()接口，然后返回进程号。

至此这几个系统调用已实现完毕。使用测试样例进行测试

试:

A screenshot of a QEMU terminal window. The title bar shows the QEMU logo and standard window controls. The terminal output consists of the following lines:

```
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.
```

根据源程序, "Child Process: Semaphore Waiting.\n"和 "Child Process: In Critical Area.\n"应打印 4 次; "Father Process: Sleeping.\n"、 "Father Process: Semaphore Posting.\n"也应打印 4 次。同时由于信号量 sem 的初值为 2, 故子进程在打印 2 次 Child Process: Semaphore Waiting.后每次再打印 Child Process: In Critical Area 前都应让父进程进行至少一次 V 操作, 也即父进程应至少打印一次 Father Process: Semaphore Posting。根据上图可知其执行结果完全符合这些条件, 故测试成功。

(2) 多线程进阶：调整框架代码中的 `gdt`、`pcb` 等进程相关的数据和代码，使得你实现的操作系统最多支持到 8 个进程 (包括内核 `idle` 进程)：

实现方法：在 `memory.h` 中，可见进程最大数量 `MAX_PCB_NUM` 的定义为 $((NR_SEGMENTS-2)/2)$ ，其中 `NR_SEGMENTS` 为 GDT 的大小。由此得知若要支持最多 8 个进程，则 $NR_SEGMENTS \geq 18$ 。为了防止越界等故障的出现，故将 `NR_SEGMENTS` 改为 18（而不是更大的数字）。这样便可实现最多 8 个进程。

(3) 理解 1.1 节中的测试程序，实现两个生产者、四个消费者的生产者消费者问题，不需要考虑进程调度，公平调度就行：

参照课本的生产者-消费者模型来编写程序。设立 4 个信号量：

- (1 `product`: 代表生产出的产品数量。初值为 0.生产者每生产一个产品则加一，消费者每消费一次则-1
- (2 `access`: 互斥信号量：用于 `product` 的互斥访问
- (3 `empty`: 指示生产者的生产。初值为 8.
- (4 `full`: 指示消费者的消费。初值为 0.

代码如下：

```
int uEntry()
{
    int ret = 0;
    sem_t product;
    sem_t access;
    sem_t empty;
    sem_t full;
    printf("Main Process: Semaphore Initializing.\n");
    ret = sem_init(&product, 0);
    ret = sem_init(&access, 1);
    ret = sem_init(&empty, 8);
    ret = sem_init(&full, 0);
    int pid = -1;
    if (ret == -1)
    {
        printf("Main Process: Semaphore Initializing Failed.\n");
        exit();
    }
}
```

首先初始化各个信号量。

```

int i;
for (i = 0; i < 6; i++)
{
    ret = fork();
    if (ret == 0)
    {
        pid = getpid();
        printf("pid===== %d\n", pid);
        //sleep(128);
        break;
    }
}

```

然后 fork 出 6 个子进程，每个子进程获得其进程编号，存在 pid 变量里。

然后开始触发生产-消费模型：

```

if (pid <= 3 && pid > 1) //producer
{
    //printf("OK here\n");
}

```

进程号为 2, 3 的子进程为生产者。（进程号为 1 的进程是主进程，也即 fork 出这 6 个子进程的进程。它会在后面 exit()掉）

```

int z;
int num = 0;

while (1)
{
    z = 0;
    while (z < 8)
    {
        num += 1;
        printf("pid: %d, producer: %d, op: produce %d\n", pid, pid, num);
        sem_wait(&empty);

        printf("pid: %d, producer: %d, op: try lock %d\n", pid, pid, num);
        sem_wait(&access);
        printf("pid: %d, producer: %d, op: locked\n", pid, pid);

        sem_post(&product);
        z++;

        sem_post(&access);
        printf("pid: %d, producer: %d, op: unlock\n", pid, pid);

        sem_post(&full);
    }
    //sleep(64);
}
}

```

然后是生产者的程序内容。生产者每次生产 8 个产品，故死循环内有一个 8 次的 while 循环。生产流程：先等待 empty 信号量满足要求；然后请求对 product 的互斥访问；然后让 product+1；解除互斥；让 full+1。期间按要求打印相关的提示。

```

else if (pid > 3) //consumer
{
    int z;
    int num = 0;

```

进程号为 4,5,6,7 的子进程为消费者。消费者程序内容：

```

while (1)
{
    z = 0;
    while (z < 4)
    {
        num++;
        sem_wait(&full);

        printf("pid: %d, consumer: %d, op: try lock %d\n", pid, pid, num);
        sem_wait(&access);
        printf("pid: %d, consumer: %d, op: locked\n", pid, pid);

        printf("pid: %d, consumer: %d, op: try consume %d\n", pid, pid, num);
        sem_wait(&product);
        z++;
        printf("pid: %d, consumer: %d, op: consumed %d\n", pid, pid, num);

        sem_post(&access);
        printf("pid: %d, consumer: %d, op: unlock\n", pid, pid);

        sem_post(&empty);
    }
    //sleep(4);
}

```

类似地，消费者每次消费 4 个产品，故死循环内有一个 4 次的 while 循环。首先等待 full 满足要求；然后请求互斥；然后消费产品；然后解除互斥；最后 empty+1。

```

    exit();
    return 0;
}

```

如果 pid 不满足上面的要求则是主进程，此时直接 exit 退出。

执行效果:

```
pid: 4, consumer: 4, op: locked
pid: 4, consumer: 4, op: try consume 130
pid: 4, consumer: 4, op: consumed 130
pid: 4, consumer: 4, op: unlock
pid: 4, consumer: 4, op: try lock 131
pid: 4, consumer: 4, op: locked
pid: 4, consumer: 4, op: try consume 131
pid: 4, consumer: 4, op: consumed 131
pid: 4, consumer: 4, op: unlock
pid: 4, consumer: 4, op: try lock 132
pid: 4, consumer: 4, op: locked
pid: 4, consumer: 4, op: try consume 132
pid: 4, consumer: 4, op: consumed 132
pid: 2, producer: 2, op: produce 181
pid: 2, producer: 2, op: try lock 181
pid: 2, producer: 2, op: locked
pid: 2, producer: 2, op: unlock
pid: 2, producer: 2, op: produce 182
pid: 2, producer: 2, op: produce 182
pid: 2, producer: 2, op: try lock 182
pid: 2, producer: 2, op: locked
pid: 2, producer: 2, op: unlock
pid: 2, producer: 2, op: produce 183
pid: 2, producer: 2, op: try lock 183
pid: 2, producer: 2, op: locked
```

```
pid: 4, consumer: 4, op: try lock 1453
pid: 4, consumer: 4, op: locked
pid: 4, consumer: 4, op: try consume 1453
pid: 4, consumer: 4, op: consumed 1453
pid: 4, consumer: 4, op: consumed 1453
pid: 4, consumer: 4, op: unlock
pid: 4, consumer: 4, op: try lock 1454
pid: 4, consumer: 4, op: locked
pid: 4, consumer: 4, op: try consume 1454
pid: 4, consumer: 4, op: consumed 1454
pid: 4, consumer: 4, op: unlock
pid: 5, consumer: 5, op: try lock 289
pid: 5, consumer: 5, op: locked
pid: 5, consumer: 5, op: try consume 289
pid: 5, consumer: 5, op: consumed 289
pid: 5, consumer: 5, op: unlock
pid: 6, consumer: 6, op: try lock 289
pid: 6, consumer: 6, op: locked
pid: 6, consumer: 6, op: try consume 289
pid: 6, consumer: 6, op: consumed 289
pid: 6, consumer: 6, op: unlock
pid: 7, consumer: 7, op: try lock 288
pid: 7, consumer: 7, op: locked
pid: 7, consumer: 7, op: try consume 288
```

```
pid: 4, consumer: 4, op: try lock 3581
pid: 2, producer: 2, op: unlock
pid: 2, producer: 2, op: produce 5011
pid: 2, producer: 2, op: try lock 5011
pid: 2, producer: 2, op: locked
pid: 2, producer: 2, op: unlock
pid: 2, producer: 2, op: produce 5012
pid: 2, producer: 2, op: try lock 5012
pid: 2, producer: 2, op: locked
pid: 2, producer: 2, op: unlock
pid: 2, producer: 2, op: produce 5013
pid: 2, producer: 2, op: try lock 5013
pid: 2, producer: 2, op: locked
pid: 2, producer: 2, op: unlock
pid: 2, producer: 2, op: produce 5014
pid: 3, producer: 3, op: try lock 715
pid: 3, producer: 3, op: locked
pid: 3, producer: 3, op: unlock
pid: 3, producer: 3, op: produce 716
pid: 4, consumer: 4, op: try lock 3580
pid: 4, consumer: 4, op: locked
pid: 4, consumer: 4, op: try consume 3580
pid: 4, consumer: 4, op: consumed 3580
pid: 4, consumer: 4, op: unlock
pid: 4, consumer: 4, op: try lock 3581
```

修改 `timehandle()` 这个时间中断处理函数, 让其打印时间
中断调度到的进程号:

