LAB4 实验报告

161220097 戚赟

一、实验要求

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

实现 SEM_INIT、SEM_POST、SEM_WAIT、SEM_DESTROY 系统调用

实现 SEM_INIT、SEM_POST、SEM_WAIT、SEM_DESTROY 系统调用，并使用以下用户程序测试

二、实验具体实现

1. 数据结构实现：

本次实现要求在实现进程的调用的情况下实现信号量对进程的控制。信号量的定义定义在 semaphore.c 文件中。

```c
#ifndef __X86_SEMAPHORE_H__
#define __X86_SEMAPHORE_H__

#define SEM_MAX_NUM 200
#define USED 1
#define FREE 0

typedef int sem_t;

struct Semaphore
{
    int used;       //shi yong
    int value;      //xin hao liang
    int pid;        //jin cheng pid
};

struct Semaphore semaphore[SEM_MAX_NUM];

void initsem();

int find_new_sem();

#endif
```

其中每个信号量有对应的进程号，used 代表是否使用过（来实现信号量的初始化分配和 destroy），value 代表信号量的值。

一共定义了信号量的数组 semaphore[SEM_MAX_NUM]

Initsem()代表初始化数组

Find_new_sem()表示从数组中寻找没有用过的来进行使用

```
#define SYS_SEM_init 250
#define SYS_SEM_post 251
#define SYS_SEM_wait 252
#define SYS_SEM_destory 253
#define SYS_sleep 300
```

自己定义的系统调用号

2. 程序执行的流程

3. 程序具体的实现流程

①信号量数组的初始化

但是这样初始化的 value 和 pid 主要是防止不必要的错误，信号量在使用之前只是看是否是已经访问过的，从而来进行判断，不能使用已 free 的信号量，这样会导致错误

```c
void initsem(void)
{
    for(int i = 0 ; i < SEM_MAX_NUM ; i++)
    {
        semaphore[i].used = FREE;
        semaphore[i].value = 0;
        semaphore[i].pid = 0;
    }
}
```

②函数的具体实现

`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`

```c
int sem_init(sem_t *sem, uint32_t value)
{
    return syscall(SYS_SEM_init, (uint32_t)sem, value, 1, 0, 0);
}
```

初始化需要将 sem 的指针地址和需要赋值的值进行传参。

```
void sys_sem_init(struct TrapFrame *tf) //chu shi hua yi ge
{
    tf->ebx += ((pcb_cur - pcb) * PROC_SIZE);
    int temp_value = tf->ecx;
    int res = find_new_sem();
    semaphore[res].value = temp_value;
    semaphore[res].pid = PID_START + pcb_cur - pcb;
    *(sem_t *)tf->ebx = res;
    tf->eax = res;  //return value
};
```

首先 tf->ebx 存放的是 sem 指针（sem_t 的类型和 int 类型无异），访问需先加上进程的基地址，通过 find_new_sem()来将寻找新的信号量，开始传参。

```
int find_new_sem()
{
    int find = -1;
    for(int i = 0 ; i < SEM_MAX_NUM ; i++)
    {
        if(semaphore[i].used == FREE)
        {
            semaphore[i].used = USED;
            find = i;
            break;
        }
    }
    return find;
}
```

信号量的初始化完毕。

下面我们先实现 P 操作

`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`

P 操作将信号量的值减少一，如果信号量的值最终减少到 0 以下，说明限制不能够继续进行改程序，需要将该程序进行阻塞。
下面讲讲遇到的问题！
这个阻塞和 lab3 中 sleep 的阻塞是不一样的，这个阻塞没有时间的限制，在 lab3 中，我用了一个 pcb_block 链表来进行了对被阻塞链表的划分，但是这并不相同，导致在 lab4 中，我将同样的放入这个链表中，导致了以外的错误！这个阻塞是只能由信号量唤醒与时间无关。
于是重新创建了一个 pcb_stop 链表来进行划分，使得这种永久停留的链表区分开来。

```
void sys_sem_wait(struct TrapFrame *tf)
{
    tf->ebx += ((pcb_cur - pcb) * PROC_SIZE);
    int sem_cur = *(sem_t*) tf->ebx;
    semaphore[sem_cur].value --;
    tf->eax = 0;
    if(semaphore[sem_cur].used == FREE)
    {
        return;
    }
```

寻找并对 value 进行操作，如果被撤销则直接进行返回。

```c
    if(semaphore[sem_cur].value < 0)        //zu shai itself
    {
        assert((pcb_cur - pcb) == 1);
        semaphore[sem_cur].pid = pcb_cur - pcb; //pid
        int pid_res = semaphore[sem_cur].pid;
        assert(pid_res == 1);
        pcb_cur->state = BLOCKED;
        struct ProcessTable * m, * n;
        if(pcb_head == pcb_cur)
        {
            m = pcb_head;
            pcb_head = pcb_head -> next;
            //assert(pcb_head != pcb);
        }
        else
        {
            m = pcb_head;
            n = m -> next; //obviously, n is not empty
            while(m -> next != NULL)
            {
                if(n == pcb_cur)//find and remove it
                {
                    m -> next = n->next;
                    break;
                }
                m = m -> next;
                n = m -> next;
            }
            if(n == NULL) assert(0);//must find one
        }
```

从当前执行的链表删除

```c
        }
        //add into BLOCKED
        if(pcb_stop == NULL)
        {
            pcb_stop = pcb_cur;
            pcb_stop -> next = NULL;
        }
        else
        {
            struct ProcessTable * m = pcb_stop;
            while(m -> next != NULL)
            {
                m = m -> next;
            }
            m -> next = pcb_cur;
            m -> next ->next = NULL;
        }
        PCB_schedule();
        //can't reach here
        assert(0);
```

加入阻塞的链表

再进行一次调度算法。

`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`

```c
void sys_sem_post(struct TrapFrame *tf)
{
    tf->ebx += ((pcb_cur - pcb) * PROC_SIZE);
    int sem_cur = *(sem_t*) tf->ebx;
    semaphore[sem_cur].value ++;
    tf->eax = 0;
    if(semaphore[sem_cur].used == FREE)
    {
        return;
    }
```

寻找信号量，值++，如果信号量被撤销，则直接返回！

```c
    if(semaphore[sem_cur].value == 0)       //guq qi
    {
        int pid_res = semaphore[sem_cur].pid;
        assert(pid_res == 1);
        assert(pcb_cur == pcb);
        pcb[pid_res].state = RUNNABLE;
        pcb[pid_res].sleepTime = 0;
        pcb[pid_res].timeCount = TIMESLICE;
```

挂起当前进程

```c
        //mov from pcb_stop
        struct ProcessTable * m, * n, *temp;
        temp = &pcb[pid_res];
        m = pcb_stop;   //stop head
        if(m == temp)
        {
            pcb_stop = pcb_stop -> next;
        }
        else{
            n = m -> next ;
            while( m -> next != NULL)
            {
                if(n == temp)
                {
                    m->next = n->next;
                    break;
                }
                m = m -> next;
                n = m -> next;
            }
            if( n == NULL) //find nothing
            {
                assert(0);
            }
        }
        assert(pcb_stop == NULL);
```

从 pcb_stop 之中删除

```
        //add into head
        m = pcb_head;
        if(pcb_head == NULL)
        {
            pcb_head = temp;
            temp -> next = NULL;
            assert(0);
        }
        else{
                n = pcb_head;
                while(n -> next != NULL)
                {
                    n = n -> next;
                }
                n -> next = temp;
                temp -> next = NULL;
        }
        assert(pcb_head != NULL);
        assert(pcb_head->next!=NULL);
        assert(pcb_head == pcb_cur);
        pcb_cur -> state = RUNNABLE;
        pcb_cur -> timeCount = TIMESLICE;
        PCB_schedule();
        assert(0);
    }
};
```

加入执行链表之中并执行

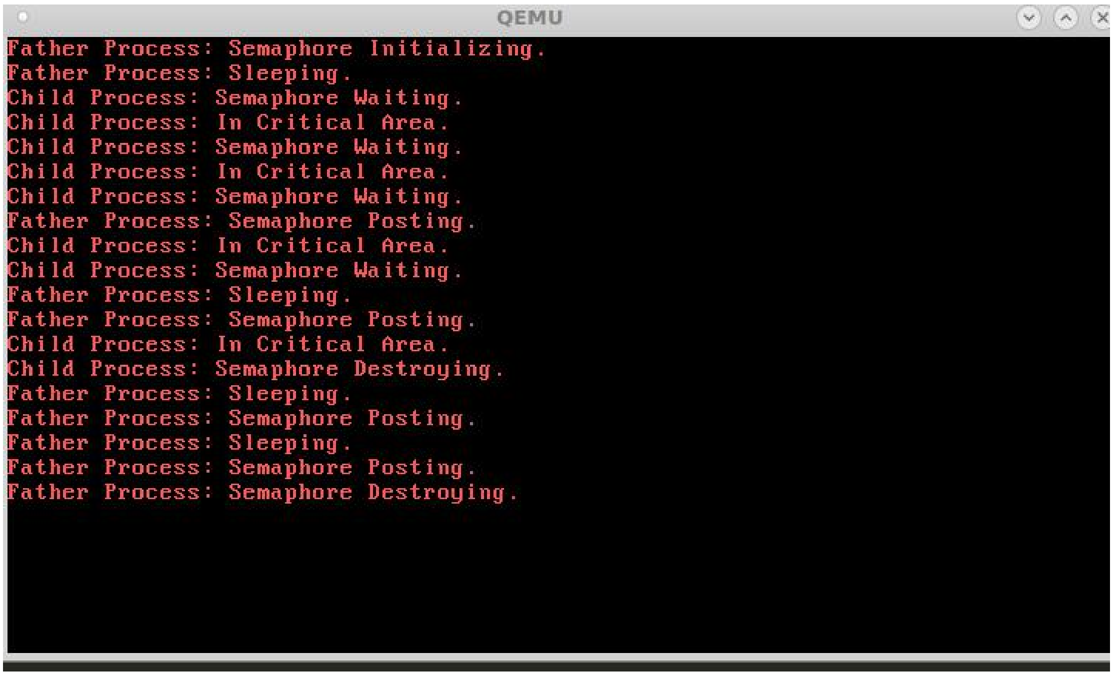
这里我做的时候有一个错误，那就是调度的时候没有将当前进程改为 RUNNABLE，失误，希望下一次逻辑能够更清楚一些。

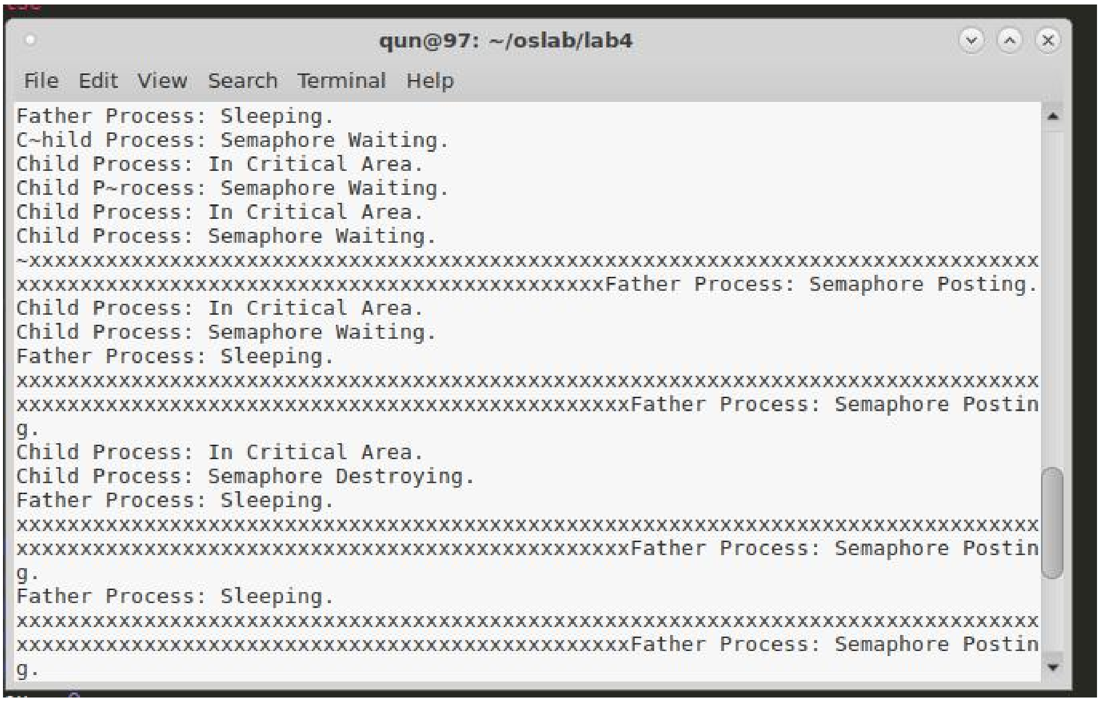
`sem_destroy` 系统调用用于销毁 `sem` 指向的信号量，销毁成功则返回 `0`，否则返回`-1`，若尚有进程阻塞在该信号量上，可带来未知错误

```
void sys_sem_destory(struct TrapFrame *tf)
{
    tf->ebx += ((pcb_cur - pcb) * PROC_SIZE);
    int sem_cur = *(sem_t*) tf->ebx;
    semaphore[sem_cur].value = -1;
    semaphore[sem_cur].pid = -1;
    semaphore[sem_cur].used = FREE;
    tf->eax = 0;
};
```

撤销

三、程序运行结果





与预期一样！