lab4 实验报告： 进程切换

姓名：王志邦
学号：161220131
日期：2018.05.27

实验目的

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

实验任务

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_destory`
2. 库：对上述系统调用进行封装
3. 用户：对上述系统调用进行测试

实验流程

本实验基于lab3进行的，所以，加载内核至内存的部分，IDT、GDT、TSS、串口、8259a的初始化任务，加载用户程序至内存的步骤，以及进程切换的任务都已经完成了。

- 用户程序的功能为测试`sem_init`、`sem_post`、`sem_wait`、`sem_destory`等系统调用，是否能够通过信号量完成对进程的同步控制，所以需要将用户进程更改为:

```
#include "lib.h"
#include "sem.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;
}
```

- 进程同步机制的实现依靠的是信号量 `semaphore`，所以内核中需要维护一个 `struct Semaphore` 结构体：
  - 其结构为：

```
struct Semaphore {
    int value;
    int waiting;
};
```

  - `value` 是信号量的值，用于控制通过 `P`， `V` 操作，控制进程的同步
  - `waiting` 是用于记录由于资源占用而等待的进程的 `pid`
- 由于，该程序仅实现简单的单信号量的进程同步，所以在我仅维护了一个信号量：`struct Semaphore semaphore;`
- **系统调用 `sem_init`、`sem_post`、`sem_wait`、`sem_destory` 的实现**
  - `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`
    - `sem_init` 系统调用过程：
      1. 由用户程序调用 `sem_init` 函数， `sem_init` 的系统调用号为规定的 **100**，调用函数时，需要将指针 `sem` 的地址与信号量的初始值 `value` 作为传入参数

```
int sem_init(int32_t *sem, int value){
        return syscall(100, (uint32_t)sem, value);
}
```

      2. 通过 `int 0x80` 陷入中断，调用 `sys_sem_init`

```
void sys_sem_init(struct TrapFrame *tf){
        tf->ebx += (current * (1 << 16));
        sem_t *sem = (sem_t*)tf->ebx;
        semaphore.value = tf->ecx;
        *sem = 0;
        tf->eax = 0;
}
```

      3. 由于分段机制与进程的共同作用，对指针 `sem` 的地址加上偏移量 `1<<16`，并对该信号量做上标记，由于我的实现中仅有一个信号量，所以，天然的 `*sem = 0`，返回值 `0` 压入 `tf->eax`
  - `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）
    - `sem_post` 系统调用过程：
      1. 由用户程序调用 `sem_post` 函数， `sem_post` 的系统调用号为规定的 **102**，调用函数时，需要将指针 `sem` 的地址作为传入参数

```
void sem_post(int32_t *sem){
        syscall(102, (uint32_t)sem, 0);
}
```

      2. 通过 `int 0x80` 陷入中断，调用 `sys_sem_post`

```
void sys_sem_post(struct TrapFrame *tf){
        if(++semaphore.value == 0){
                int wake = semaphore.waiting;
                assert(wake == 1);
                pcb[wake].state = RUNNABLE;
                pcb[wake].timeCount = 16;
                pcb[current].state = RUNNABLE;
                //pcb[current].timeCount = 16;
                schedule();
        }
}
```

3. 由于仅有一个信号量，不需要根据信号量的索引号，去寻找此次操作的信号量，仅要对 `semaphore.value` 进行自增操作，判断 `semaphore.value` 为 `0` 时，将 `semaphore.waiting` 进程唤醒，修改其 `state` 为 `RUNNABLE`，进行进程调度

- `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
  - `sem_wait` 系统调用过程：
    1. 由用户程序调用 `sem_wait` 函数， `sem_wait` 的系统调用号为规定的 **101**，调用函数时，需要将指针 `sem` 的地址作为传入参数

```
void sem_wait(int32_t *sem){
        syscall(101, (uint32_t)sem, 0);
}
```

    2. 通过 `int 0x80` 陷入中断，调用 `sys_sem_wait`

```
void sys_sem_wait(struct TrapFrame *tf){
        if(--semaphore.value < 0){
                pcb[current].state = BLOCKED;
                semaphore.waiting = current;
                current = -1;
                schedule();
        }
}
```

    3. 由于仅有一个信号量，不需要根据信号量的索引号，去寻找此次操作的信号量，仅要对 `semaphore.value` 进行自减操作，判断 `semaphore.value` 小于 `0` 时，将当前进程状态修改为 `BLOCKED`，并将当前进程的 `pid` 存入 `semaphore.waiting`，进行进程调度

- `sem_destory`
  - `sem_destory` 系统调用用于销毁 `sem` 指向的信号量，若当前进程为子进程，则尚有进程阻塞在该信号量上，不可销毁
  - 1. 由用户程序调用 `sem_destory` 函数， `sem_destory` 的系统调用号为规定的 **103**，调用函数时，需要将指针 `sem` 的地址作为传入参数

```
void sem_destroy(int32_t *sem){
        syscall(103, (uint32_t)sem, 0);
}
```

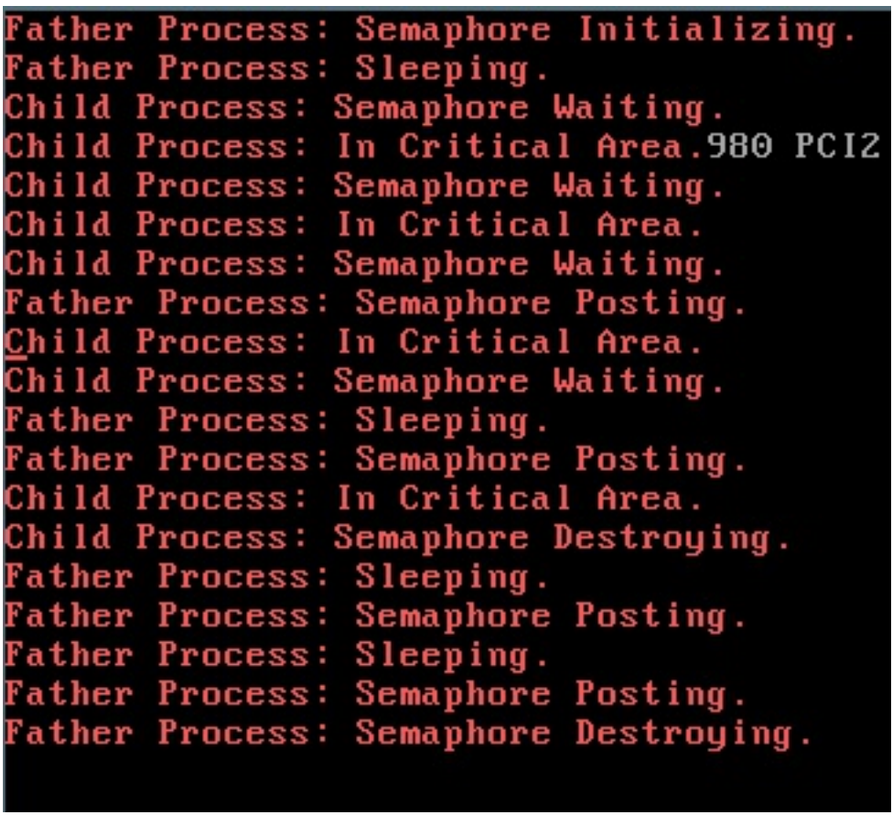
2. 通过 `int 0x80` 陷入中断，调用 `sys_sem_destory`

```
void sys_sem_destory(struct TrapFrame *tf){
        if(pcb[current].pid == 521)
                return;
        else if(pcb[current].pid == 520){
                tf->ebx += (current * (1 << 16));
                sem_t *sem = (sem_t*)tf->ebx;
                *sem = -1;
        }
```

```
}
```

> 由于程序十分简单，运行过程中，几个系统调用并不会产生失败的结果，所以，我将几个函数从int型有返回值的函数，更改为了void型无返回值的函数

运行结果



实验过程中遇到的一些问题

> 由于，开始的时候，没有在结构体中记录被阻塞的进程识别号，在进程被`sem_wait`阻塞以后，无法知道应该使哪个进程开始运行，在将信号量结构中加入`waiting`，这样就可以在`P`操作之后，恢复被阻塞的进程

> 在`destory`函数的实现上，由于没有考虑到，父子进程全部依赖信号量`semaphore`，在`sem_destory`时，直接在第一次直接将该信号量销毁，导致，程序在`Child Process: Semaphore Destorying`之后停止运行，在将程序中加入对子进程的判断后：`if(pcb[current].pid == 521) return;`程序可以正常进行了