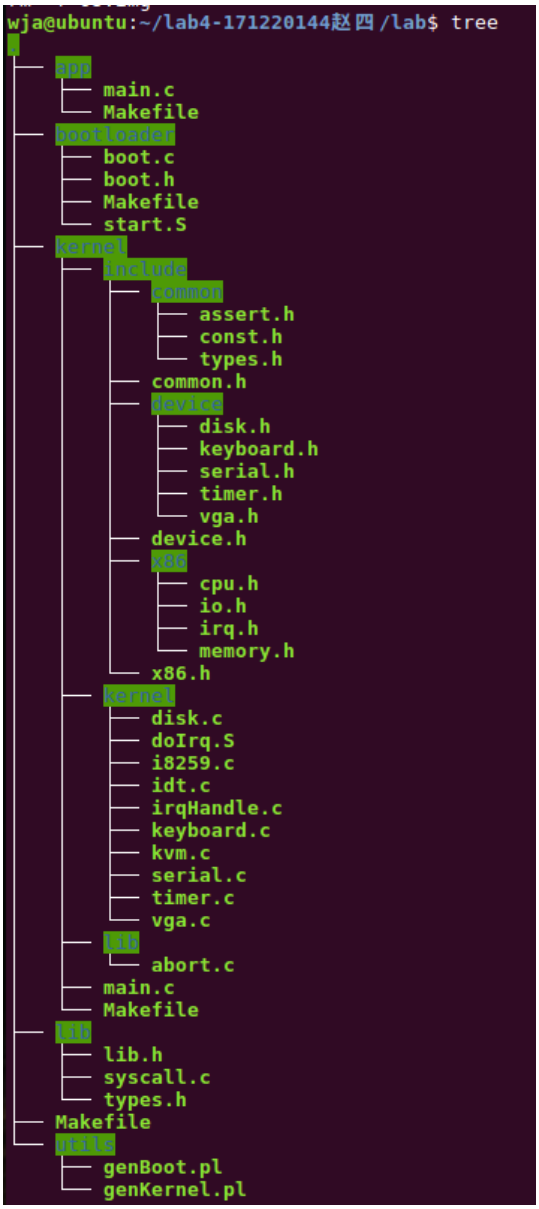
Lab4 进程同步

vigorweijia@foxmail.com

一、代码框架及其解读



相较于上一个实验，本次的框架基本相同，在上一次实验的基础代码上加入了对信号量的操作。包括信号量的创建、增减以及删除。还需要增加获得进程号的系统调用以及支持8个进程。

由于信号量真实保存在内核中，而不是保存在用户栈下，所以操作信号量的时候是取其指针。信号量操作函数定义在lib/syscall.c种，是库函数的一部分。他会进一步调用系统调用syscall，然后跳转到kernel/kernel/doirq.S下的irqSyscall以及asmDoIrq然后跳转到irqHandle.c中的irqHandle函数，检查中断号发现是系统调用，于是调用其处理函数syscallHandle，进一步根据寄存器eax的值判断是何种系统调用。

如果是信号量的相关操作，会调用syscallSem函数。而框架代码中没有获取调用号的操作，所以还需要自己创建一个和获取信号量相关的宏SYS_PID以及其处理函数。对于信号量相关操作，会根据ecx进一步判断具体是哪种信号量的操作。包括Init（初始化）、Wait（信号量P操作，表示消耗资源）、Post（信号量V操作，增加/释放一个资源）、Destroy（释放信号量）。

在信号量系统调用函数上面有一个系统调用syscallReadStdIn。里面定义了一个与信号量类似的结构dev，即设备结构体。

```
#define MAX_SEM_NUM 4

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

#define MAX_DEV_NUM 4

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

因为多个进程可能会同时占用设备进行输入输出，但是每个时刻一个设备只能被一个进程占用，所以进程使用设备也是互斥的，一个进程抢夺到了设备，另一个进程必须等待，这个机制和信号量的作用基本是一样的，所以信号量操作实现可以参考设备操作的实现。

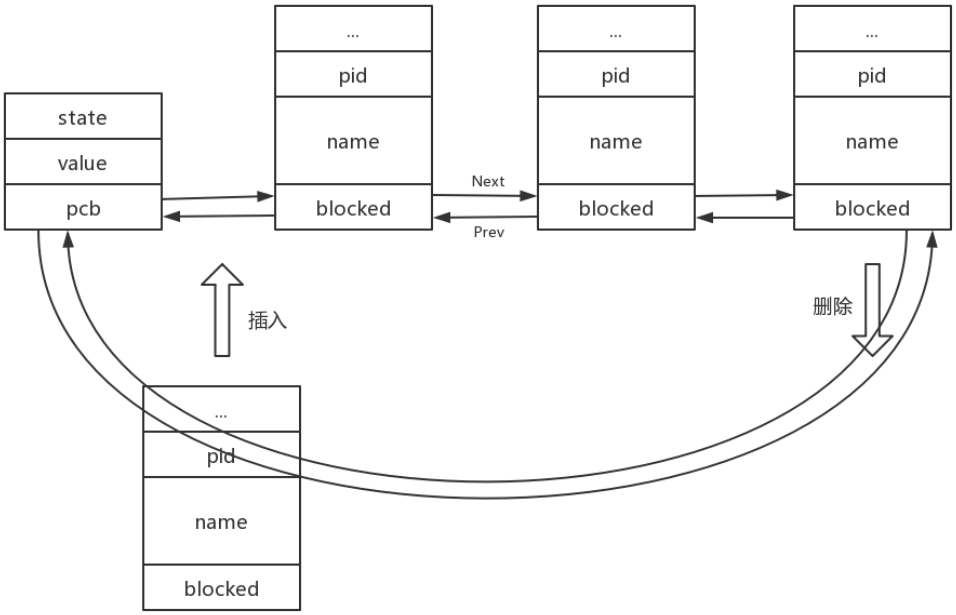
二、思路

信号量操作

信号量的初始化（SemInit），创建信号量的时候需要对其进行初始化，赋予其资源的数量。首先遍历信号量表，找到一个空闲的位置。信号量初始化的值保存在edx中。初始化的时候，由于没有任何进程需要等待该信号量对应的资源，因此，挂起链表（双向环状）的next和prev都指向了自己。创建了信号量，返回信号量对应的表项，保存在eax中。下一次使用信号量即可根据索引来找到信号量表中的对应位置。因此，我们在用户进程下创建的信号量，其整数值实际上代表该信号量在信号量表上的索引。

接着是信号量的等待操作（P操作）。该操作需要消耗一个资源，所以sem[i].value--（i的值保存在edx）。这些可以在系统调用的时候知晓。如果此时该资源的值小于0了，需要等待，于是阻塞该进程。即把该进程挂起在信号量的链表上，然后调用int $0x20指令切换进程。为了和SLEEP的阻塞效果区别开来，此时sleeptime修改为-1。

难点在“挂起到链表“上。链表结构如下，进程控制块中写好了对应的链表结构。



为了节省空间，链表节点就只有两个值，一个next和一个prev，分别指向前面后后面的进程控制块。得到了进程控制块的blocked变量的地址后，可根据这个地址减36得到pid的地址，进而获得进程的pid。

信号量的post操作（对应信号量V操作），会释放资源。若value<=0，说明此时有进程正在等待资源，为了公平起见，把这个资源分配给链表末尾的进程。把对应pid进程的状态改为RUNNABLE等待调度。

信号量的删除操作。直接把state改为0表示空闲。倘若此时仍有进程挂起在该信号量上，会报错。

getpid

该操作完全仿照其他系统调用，如exit、fork等。处理函数命名为syscallGetPid。然后把相应进程控制块返回值，即eax，改为pid即可。

样例

原始样例无需修改。

要求要有两个生产者、四个消费者、一个父进程以及一个idle进程，每个进程占用两个段表项，他们一共占用16个段表项。而TSS还会占用两个段表项。所以一共需要18个段表项。QEMU模仿了i386架构，寻址空间为4G，而一个段的偏移步长为0x100000，所以只需要修改段表最大项宏MAX_PCB_NUM，而这个宏和NR_SEGMENTS宏有关。

三、代码完善

信号量操作的代码如思路所示。

getpid操作代码如思路所示，将NR_SEGMENTS改为大于等于18的数。

修改测试样例。

父进程需要一个循环来执行6次fork。若返回值为0，表示子进程，终止循环，否者不只产生6个进程。

每个生产者生产8个产品，每个消费者生产4个产品。

生产者在进入临界区之前，打印produce，然后消耗empty资源，这个时候尝试访问buffer，buffer必须互斥访问，所以打印try lock，然后消耗互斥访问buffer的资源。打印locked表示正在访问buffer，把产品放入buffer。接着释放互斥资源，打印unlock。最后增加full信号量，表示生产了一个产品，此时执行sleep操作唤醒另一个进程，如此循环8次。

消费者在进入临界区之前，打印try consume表示准备消费，表示需要从buffer取。然后消耗full资源，因为要保证buffer不为空。进入buffer前要消耗互斥访问的资源，所以打印try lock。访问buffer时候打印locked，表示从buffer中取出资源。然后释放互斥资源，打印unlock，再增加empty信号量，表示空闲空间多一个。最后打印consume表示消耗产品。消耗完后执行sleep切换到其他进程。如此循环4次。

empty的初始值设为一个较大的数，因为buffer的空闲空间可以视作无限大，其实empty这个信号量是可有可无的。full信号量初始值为0，表示buffer为空。mutex初始值为1，表示对buffer的互斥访问。

注意，生产与消费产品的过程是互不干涉的，即可以被随便打断，在实际情况中它们占用时间更多。因此需要把这两个操作放在临界区以外，否则执行效率会很低。

所有操作执行完后执行exit显式终止该进程。

```
int uEntry(void) {

    sem_t empty;
    sem_t full;
    sem_t mutex;

    int ret1,ret2,ret3;

    ret1 = sem_init(&empty, 100);
    ret2 = sem_init(&full, 0);
    ret3 = sem_init(&mutex, 1);

    if(ret1 == -1 || ret2 == -1 || ret3 == -1) {
        printf("Father Process: Semaphore Initializing
Failed.\n");
        exit();
    }

    for(int i = 0; i < 6; i++) {
        int ret = fork();
        if(ret == 0) break;
    }

    int pid = getpid();

    if(pid >= 2 && pid <= 3) {
        for(int i = 1; i <= 8; i++) {
            printf("pid %d, producer %d, produce, product
%d\n",pid,pid-1,i);
            sem_wait(&empty);
            printf("pid %d, producer %d, try lock\n",pid,pid-1);
            sem_wait(&mutex);
            printf("pid %d, producer %d, locked\n",pid,pid-1);
            sem_post(&mutex);
            printf("pid %d, producer %d, unlock\n",pid,pid-1);
            sem_post(&full);
            sleep(128);
        }
```

```
    }
    else if(pid >= 4 && pid <= 7){
        for(int i = 1; i <= 4; i++) {
            printf("pid %d, consumer %d, try consume, product
%d\n",pid,pid-3,i);
            sem_wait(&full);
            printf("pid %d, consumer %d, try lock\n",pid,pid-3);
            sem_wait(&mutex);
            printf("pid %d, consumer %d, locked\n",pid,pid-3);
            sem_post(&mutex);
            printf("pid %d, consumer %d, unlock\n",pid,pid-3);
            sem_post(&empty);
            printf("pid %d, consumer %d, consumed, product
%d\n",pid,pid-3,i);
            sleep(128);
        }
    }

    //sem_destroy(&empty);
    //sem_destroy(&full);
    //sem_destroy(&mutex);
    exit();
    return 0;
}
```

初始的父进程不做为生产者和消费者，只fork出新的进程。若为子进程，则根据其pid执行相应的功能。为了公平调度，在每轮生产、消费之后，需要进行sleep操作切换进程。

在我的样例种，程序的末尾不能有sem_destroy。父进程或某一个子进程会先于其他进程结束，然后后释放调信号量，导致其他进程对信号量进行操作的时候出错。

在return之前还需要显式调用exit()。

QEMU窗口能够显示的内容有限，于是重定向stdout（准确的说是增加），让其输出到串口。

```
void syscallWriteStdOut(struct StackFrame *sf) {
    //...
    for (i = 0; i < size; i++) {
        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"
(str+i));
        putChar(character);
        //...
    }
```

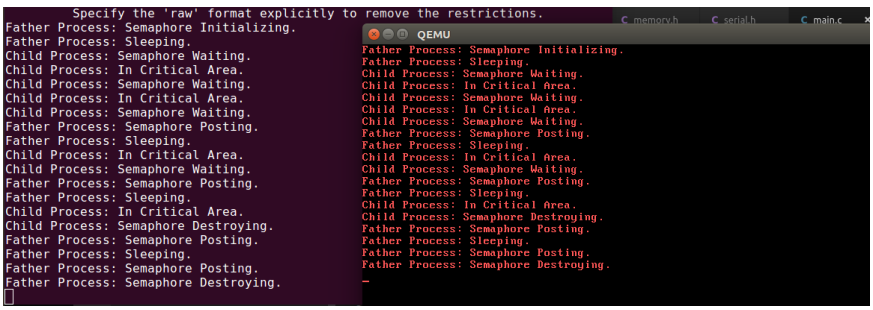
若直接putChar(str[i])，会出现乱码的情况。

本次实验更好的实现方式是采用线程之间的信号量，因为进程并不会共享某一变量（缓冲区）的值。

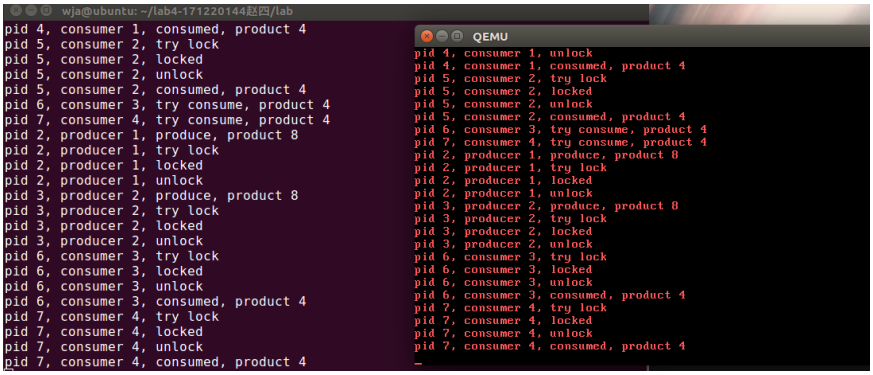
四、运行结果及分析

原始版本

原测试样例中，资源的初始值为2，所以子进程在消耗完2个资源后应该被阻塞，此时唤醒父进程进行生产，生产完后再次唤醒子进程。子进程消费完后再次等待被挂起，此时唤醒父进程生产。生产完后唤醒子进程，消耗完后循环结束，唤醒父进程。父进程生产完最后产品后执行destroy操作。



自己编写的样例，改到terminal和QEMU同时输出



完整输出

由于在每次生产/消费有sleep操作，表示切换进程，公平调度，所以执行的时候不会一个进程执行一长段时间。