中山大学 SUN YAT-SEN UNIVERSITY

操作系统原理实验报告

实验八：进程同步与信号量机制

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

一、 实验目的

- 学习信号量机制的原理，掌握其实现方法。
- 利用信号量机制，实现进程互斥和同步，理解并发进程的同步与互斥原理。
- 扩展MyOS，实现信号量机制。

二、 实验要求

如果内核实现了信号量机制相关的系统调用，并在c库中封装相关的系统调用，那么，我们的c语言就也可以实现多进程同步的应用程序了。

例如：利用进程控制操作，父进程f创建二个子进程s和d，大儿子进程s反复向父进程f祝福，小儿子进程d反复向父进程送水果（每次一个苹果或其他水果），当二个子进程分别将一个祝福写到共享数据a和一个水果放进果盘后，父进程才去享受：从数组a收取出一个祝福和吃一个水果，如此反复进行。

1. 适当增加操作时延，观察和捕捉RC问题，截屏并说明。
2. 利用信号量和PV操作协调进程的并发

三、 实验环境

具体环境选择原因已在实验一报告中说明。

- Windows 10系统 + Ubuntu 18.04(LTS)子系统
- gcc 7.3.0 + nasm 2.13.02 + GNU ld (Binutils) 2.3.0
- GNU Make 4.1
- Oracle VM VirtualBox 6.0.6
- Bochs 2.6.9
- Sublime Text 3 + Visual Studio Code 1.33.1

虚拟机配置：内存4M，1.44M虚拟软盘引导，1.44M虚拟硬盘。

四、 实验方案

本次实验继续沿用实验六保护模式的操作系统。

本节所讲的内容基本都在`semaphore.h`头文件中实现。

1. 信号量的定义

信号量即一个整数和一个指针组成的结构体，整数是信号量的值，指针指向该信号量的进程队列。

```
typedef struct Semaphore {
    int val;
    bool used;
    process* head;
} Semaphore;
```

由于有多个信号量，为方便管理，故添加了一个布尔类型used，用以判断该信号量是否被使用。进而可以用一个数组来存储所有的信号量。

```
Semaphore sem_list[N_SEMAPHORE];
```

2. 初始化信号量

这部分内容实施在sem_init中，即对每一信号量的used位置0，并将阻塞队列置空。

3. 获取及释放信号量

对信号量数组进行遍历，若某一信号量未被使用，则进行初始化，并返回其索引。其中参数val为信号量的初始值。

```
int do_getsem(int val) {
    for(int i = 0; i < N_SEMAPHORE; ++i) {
        if(!sem_list[i].used) {
            sem_list[i].val = val;
            sem_list[i].used = true;
            return i;
        }
    }
    return -1;
}
```

释放信号量则更为简单，直接将其used位置0即可，同时头指针指空。

```
void do_freesem(int sem_id) {
    sem_list[sem_id].used = false;
    sem_list[sem_id].head = NULL;
}
```

4. P操作

P操作或者称semWait，即对信号量的值减1。如果信号量的值**小于0**，则阻塞当前进程。

为实现阻塞队列，我在原process结构体中添加了process* next的成员，用于构建阻塞链表。当当前进程需要被阻塞时，将当前进程的链表指针指向该信号量阻塞队列的头部，然后

将信号量的阻塞队列指针指向当前进程。这里的调度策略是类似栈的FILO策略，因为每次都插入在链表头部，这可以大大避免遍历链表的开销，但可能造成某些进程的执行次数较少。

注意要保证P操作的原子性，这里采用了不那么优美的关中断方法，同样能实现操作。

```
void do_sem_p(int sem_id) { // sem_wait
    disable();
    sem_list[sem_id].val--;
    if (sem_list[sem_id].val < 0) {
        curr_proc->next = sem_list[sem_id].head;
        sem_list[sem_id].head = curr_proc;
        do_wait();
    }
    enable();
}
```

5. V操作

V操作或者称semSignal，即对信号量的值加1。如果信号量的值**小于等于0**，则唤醒一阻塞进程。

唤醒进程直接从阻塞队列头部获取即可，然后将队列指针向后移一位，即实现阻塞队列的缩减。

同样要注意保证V操作的原子性，需要进行关中断。

```
void do_sem_v(int sem_id) { // sem_signal
    disable();
    sem_list[sem_id].val++;
    if (sem_list[sem_id].val <= 0) {
        int pid = sem_list[sem_id].head->pid;
        sem_list[sem_id].head = curr_proc->next;
        wakeup(pid);
    }
    enable();
}
```

6. 互斥锁

由于互斥锁(mutex)相当于是一个二元信号量，故在semaphore.h中添加几行代码就顺便实现了。

```
int mutex_get() {
    return do_getsem(1);
}

void mutex_lock(int lock_id) {
    do_sem_p(lock_id);
```

```
}

void mutex_unlock(int lock_id) {
    do_sem_v(lock_id);
}
```

7. 系统调用

为使**用户态(ring 3)程序**能够正常调用信号量操作，类似之前几次实验，同样将信号量操作封装在系统中断`int 0x80`中。

在`api.h`头文件中提供了四个信号量操作的用户态函数入口，即`get_sem`、`sem_wait`、`sem_signal`、`free_sem`。用户程序只需包含该头文件即可实现调用，由于只包含`int`语句，而具体实施在内核中，既能缩减用户程序大小，又能保证内核代码的安全性，实现了封装的效果。

目前实现的系统调用及中断号请见附录B。

五、 实验结果

本次实验实现了三个不同的用户程序，用以说明RC问题及信号量的控制机制。

1. 银行存取款

代码实现在`bank.c`中，这里截取核心代码。通过声明一个互斥信号量（初值为1）实现父亲的存款及儿子的提款。父亲一共存款100次，每次存款10元；儿子一共提款50次，每次提款20元。初始账户余额为1000元，故经过存取款后总余额应该保持不变。

```
#include "stdio.h"
#include "api.h"

int sem;
int bank_account = 1000;

void main() {
    sem = get_sem(1);
    int pid = fork();
    if (pid == -1) {
        printf("error in fork!");
        exit(-1);
    }
    if (pid) { // parent
        for (int i = 0; i < 100; ++i) {
            sem_wait(sem);
            bank_account += 10;
            printf("Father: deposite $10 Account: $%d\n", bank_account);
            sem_signal(sem);
        }
```

```
    } else { // child
        for (int i = 0; i < 50; ++i) {
            sem_wait(sem);
            bank_account -= 20;
            printf("Child: withdraw $20 Account: $%d\n", bank_account);
            sem_signal(sem);
        }
        exit(0);
    }
    wait(); // parent
    return;
}
```

实验结果如图1所示。其中上图为存取款过程中的截屏，下图为最终的结果。可以看出父子的存取款过程是交替进行的，但最终依然能够保持总额不变，说明信号量实现的正确性。

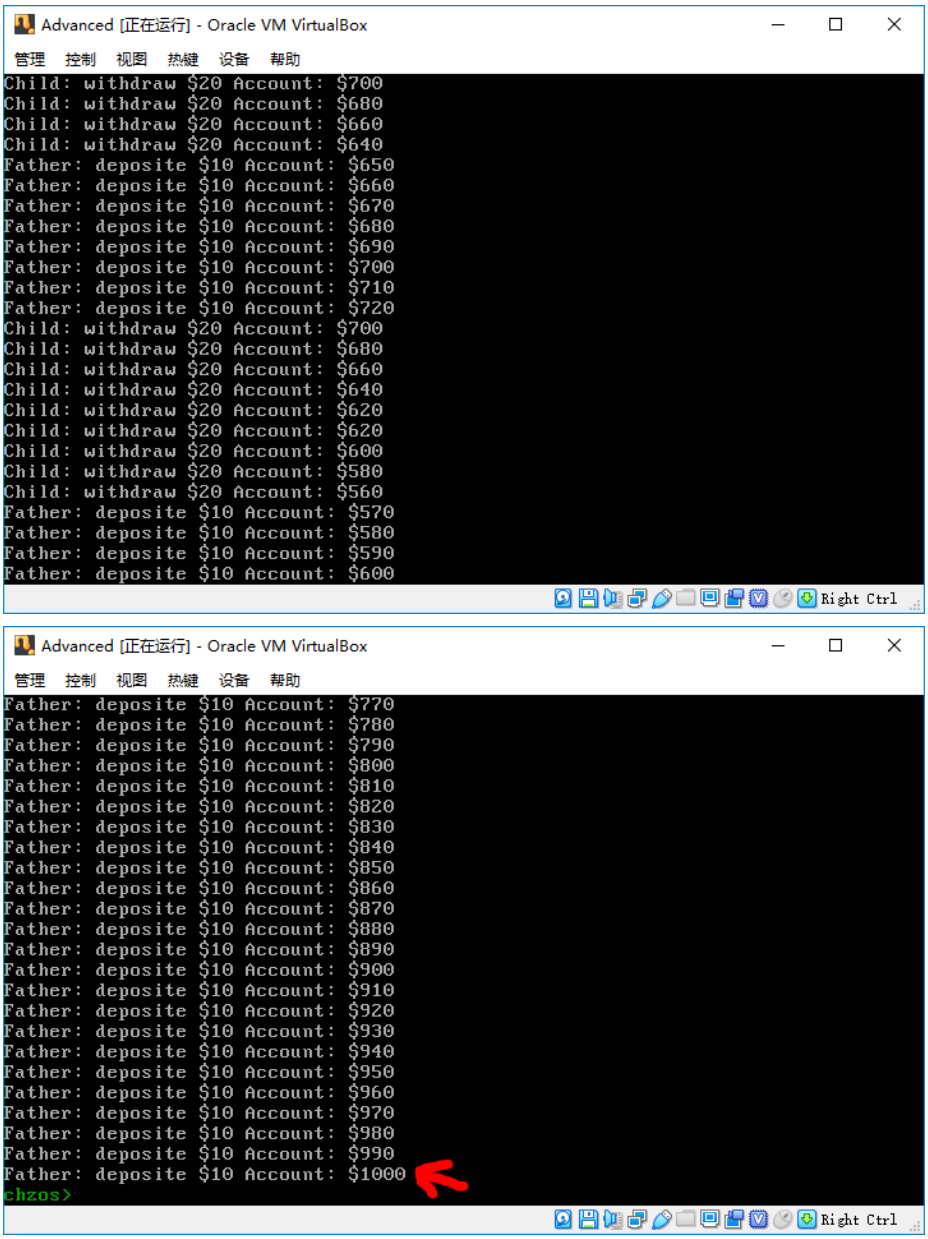

图 1: 银行存取款结果

2. 父子祝福水果

这是本实验要求中要完成的程序，代码如下。儿子依次向父亲送上苹果、梨和香蕉，如此往复。

注意这里对老师课件上的程序进行了修改，**原程序是存在bug的！** 原程序中只使用了一个信号量，而这里我采用了四个信号量，以确保程序的正确性，具体原因下面会说明。

```
#include "stdio.h"
#include "api.h"
```

```
enum FRUITS {
    NONE = 0,
    APPLE,
    PEAR,
    BANANA
} fruit_plate;

char words[100];
int fruit_index;

void put_words(char* w)
{
    strcpy(words,w);
}

void put_fruit()
{
    fruit_index = (fruit_index + 1) % 4;
    if (fruit_index == 0)
        fruit_index++;
    fruit_plate = fruit_index;
}

int sem_word, sem_fruit, sem_word_full, sem_fruit_full;

void main(){
    sem_word = get_sem(0);
    sem_fruit = get_sem(0);
    sem_word_full = get_sem(0);
    sem_fruit_full = get_sem(0);
    fruit_plate = fruit_index = 0;
    if (fork()){
        while(1){
            sem_wait(sem_word); // p0
            sem_wait(sem_fruit); // p1
            char fruit_name[10];
            if (fruit_plate == APPLE)
                strcpy(fruit_name,"APPLE");
            else if (fruit_plate == PEAR)
                strcpy(fruit_name,"PEAR");
            else if (fruit_plate == BANANA)
                strcpy(fruit_name,"BANANA");
            printf("%s %s!\n",words,fruit_name);
            sem_signal(sem_word_full); // v2
            fruit_plate = 0;
```

```
            sem_signal(sem_fruit_full); // v3
        }
    } else if (fork()) {
        while(1){
            put_words("Father will live forever!");
            sem_signal(sem_word); // v0
            sem_wait(sem_word_full); // p2
        }
    } else {
        while(1){
            put_fruit();
            sem_signal(sem_fruit); // v1
            sem_wait(sem_fruit_full); // p3
        }
    }
}
```

如果只采用一个信号量，即祝福和放水果都用同一个，这并无法保证父亲每次**恰好收到一份祝福和一个水果**时就输出，可能会出现两份祝福而没有水果，或两个水果而没有祝福的情况，因为这都符合信号量增加到0，取消阻塞的机制。

如果将祝福和放水果的信号量分离，即采用两个信号量，也同样是有问题的。因为儿子并不知道父亲已经将祝福和水果拿走，因此在父亲没拿走这些物品之前，儿子就已经把新的放了上去，父亲拿走的时候就拿空了，下一次父亲就会以为儿子还没有放。体现在程序中即父进程对`words`和`fruit_name`输出后，需要更新`fruit_plate`为0，但在更新之前子进程实际上已经完成了`put_fruit`操作，并且不断循环。此时`fruit_plate=0`覆盖掉了子进程放上去的水果，从而导致RC问题。如图2所示，水果出现的次序并不是依照`APPLE,PEAR,BANANA`这样，而更多是`APPLE`，这正是因为`fruit_plate`被清空了，只能从0开始计数，进而导致意料之外的答案。

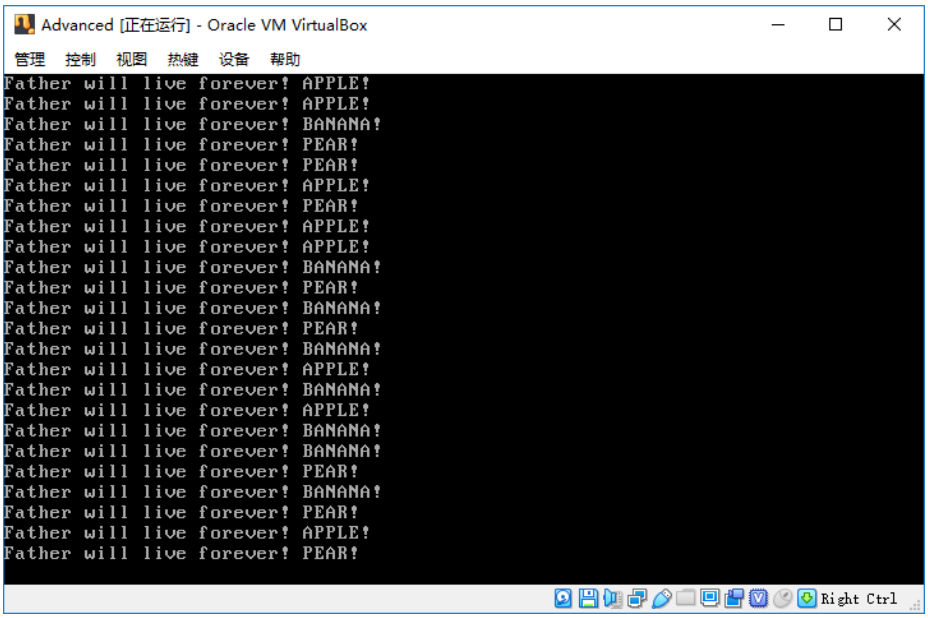

图 2: 祝福水果RC问题

而正确的实现方式应该如我的代码所示，采用四个信号量。 `sem_word`和`sem_fruit`用来判断盘中是否有东西，儿子需要放了祝福或水果后父亲才能享用，这也是老师代码中所体现的。但是老师的代码中忽视了还有另一个同步关系，即父亲享用完儿子才能放新的祝福或水果。故需要`sem_word_full`和`sem_fruit_full`用来判断盘中是否满了，如果满了则不应该放置新的物品，防止新的东西被覆盖清除导致错误。

实验结果如图3所示，可以清晰地看到水果出现的次序严格依照`APPLE`,`PEAR`,`BANANA`出现，解决了RC问题，确保了程序正确性。

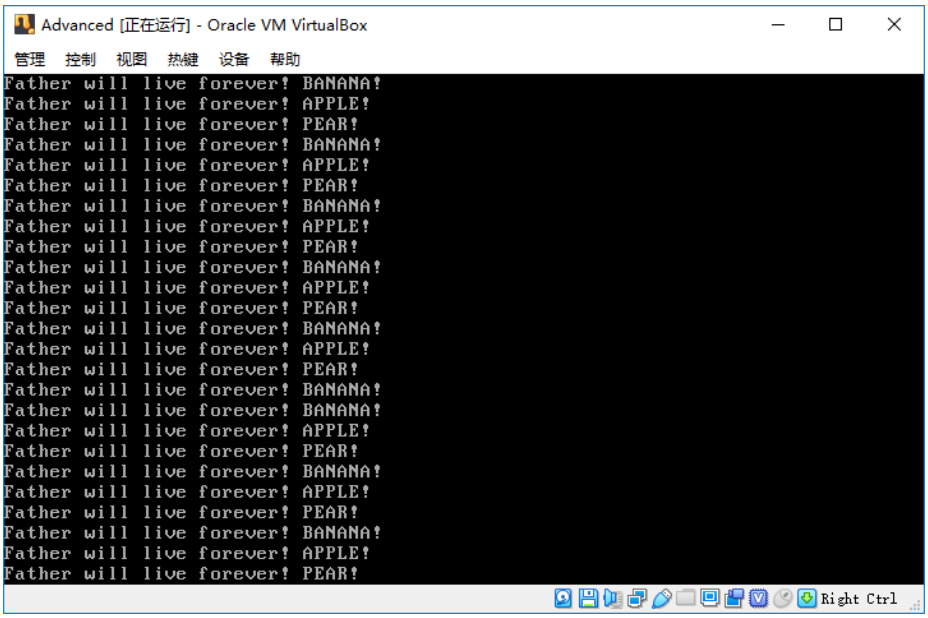

图 3: 祝福水果正确结果。注意由于动态截图的问题，故中间会出现两行同样的输出，但实际上这是输出太快而导致的截图暂留，原始输出结果是正确无误的!

3. 生产者消费者模型

本次实验还用信号量实现了生产者消费者模型，如下代码所示。其中信号量mutex用来保证互斥性，信号量n用来保证同步关系。

```
#include "stdio.h"
#include "api.h"

int tot;
int mutex;
int n;

void main() {
    tot = 0;
    mutex = get_sem(1);
    n = get_sem(0);
    int pid = fork();
    if (pid == -1) {
        printf("error in fork!");
        exit(-1);
    }
    if (pid) {
        while (1) { // Producer
            sem_wait(mutex);
            printf("Produce %d\n", ++tot);
            sem_signal(mutex);
```

```
            sem_signal(n);
        }
    } else {
        while (1) { // Consumer
            sem_wait(n);
            sem_wait(mutex);
            printf("Consume %d\n", tot--);
            sem_signal(mutex);
        }
    }
}
```

实验结果如图4所示，可以看出只有生产者产生了某一序号的物品，消费者才会对其进行消费，如此持续。

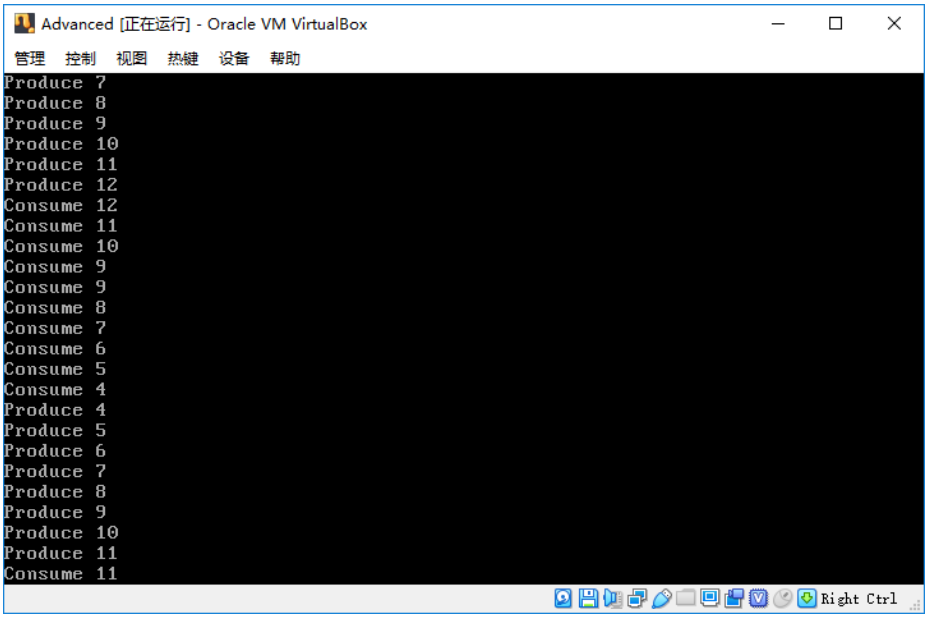

图 4: 生产者消费者模型结果。注意同样存在动态截图导致两行重复的问题，但结果是正确的!

同上一个祝福水果实验，实验结果都说明我的操作系统能够正确解决多进程、多信号量的情况，具有充分的灵活性和鲁棒性。

六、 实验总结

本次实验有了之前实验的基础，相对来讲比较简单，只需对照着信号量的原理，将四个基本操作实现一遍即可。

但实现过程中还是因为自己的疏忽出现了一些小问题。如一开始以为信号量只需维护一个阻塞进程即可，在做银行存取款实验时没有出现任何问题，但到了父子祝福实验就始终做不对结果。原因正是因为父子祝福实验涉及到三个进程，而仅仅维护一个阻塞进程就可能出现bug。

发现问题之后对相应的进程控制块进行修改，并在信号量操作中对阻塞队列进行维护，就可以确保在多进程环境下程序运行也不会出错了。

另一点则是在编写用户程序时的疏忽—误把信号量开为局部变量，导致信号量的值总是不如期望，逐行调试了很久才发现了这个问题。不管是进程号、信号量、互斥锁等等，这些涉及到多进程协作的变量，就应该开设在全局，这样才有办法让各个进程共享使用，否则当新的进程产生后，它就不知道去哪里找对应的数据了，因为进程的克隆并没有拷贝所有的变量/内存内容。

总的来说，实现操作系统的时候还是应该更小心点。对于操作系统来说，一个小的bug都很可能是致命的，这点尤为注意。

七、 参考资料

1. OS Development Series, http://www.brokenthorn.com/Resources/OSDevIndex.html
2. Roll your own toy UNIX-clone OS, http://www.jamesmolloy.co.uk/tutorial_html/
3. The little book about OS development, http://littleosbook.github.io/
4. Writing a Simple Operating System from Scratch, http://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf
5. Intel® 64 and IA-32 Architectures Software Developer's Manual
6. UCore OS Lab, https://github.com/chyyuu/ucore_os_lab
7. CMU CS 15-410, Operating System Design and Implementation, https://www.cs.cmu.edu/~410/
8. 李忠，王晓波，余洁，《x86汇编语言-从实模式到保护模式》，电子工业出版社，2013

附录 A. 程序清单

1. 内核核心代码

序号	文件	描述
1	bootloader.asm	主引导程序
2	kernel_entry.asm	内核汇编入口程序
3	kernel.c	内核C入口程序
4	Makefile	自动编译指令文件
5	bootflpy.img	引导程序/内核软盘
6	mydisk.hdd	虚拟硬盘
7	bochsrc.bxrc	Bochs配置文件

2. 内核头文件

序号	文件	描述
1	disk_load.inc	BIOS读取磁盘
2	show.inc	常用汇编字符显示
3	gdt.inc	汇编全局描述符表
4	gdt.h	C全局描述符表
5	idt.h	中断描述符表
6	hal.h	硬件抽象层
6.1	pic.h	可编程中断控制器
6.2	pit.h	可编程区间计时器
6.3	keyboard.h	键盘处理
6.4	tss.h	任务状态段
6.5	ide.h	硬盘读取
7	io.h	I/O编程
8	exception.h	异常处理
9	syscall.h	系统调用
10	task.h	多进程设施
11	user.h	用户程序处理
12	terminal.h	Shell
13	scancode.h	扫描码
14	stdio.h	标准输入输出
15	string.h	字符串处理
16	elf.h	ELF文件处理
17	api.h	进程管理API
18	semaphore.h	信号量机制

3. 用户程序

用户程序都放置在usr文件夹中。

序号	文件	描述
1-4	`prgX.asm`	飞翔字符用户程序
5	`box.asm`	画框用户程序
6	`sys_test.asm`	系统中断测试
7	`fork_test.c`	进程分支测试
8	`fork2.c`	进程多分支测试
9	`bank.c`	银行存取款测试
10	`fruit.c`	父子祝福水果测试
11	`prod_cons.c`	消费者生产者模型测试

附录 B. 系统调用清单

功能号	功能
0	输出OS Logo
1	睡眠100ms
10	`fork`
11	`wait`
12	`exit`
13	`get_pid`
20	`get_sem`
21	`sem_wait`
22	`sem_signal`
23	`free_sem`
100	返回内核Shell