实验八：实现信号量机制与应用

19335174施天予

一、实验题目

实现信号量机制与应用

二、实验目的

1. 理解基于信号量的进程/线程同步方法
2. 掌握内核实现信号量的方法
3. 掌握信号量的应用方法
4. 实现C库封装信号量相关系统调用

三、实验要求

1. 学习信号量机制原理
2. 扩展内核，设计实现一种计数信号量。提供信号量申请、初始化和p操作与v操作等功能的系统调用
3. 修改扩展C库，封装信号量相关的系统调用操作
4. 设计一个多线程同步应用的用户程序，展示你的多线程模型的应用效果
5. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

四、实验内容

1. 在没有互斥机制时，实际运行父子存款取款问题会产生竞态，设法截屏验证此事
2. 修改内核代码，增加信号量结构类型并定义一组信号量，组成信号量数组等数据结构。提供信号量申请初始化、释放和p操作与v操作等功能的系统调用
3. 修改扩展C库，封装信号量相关的系统调用操作
4. 设计一个多线程同步应用的用户程序，展示你的多线程模型的应用效果
   应用1：用你的C库，编程解决多线程问题。进程创建2个线程，分别代表父亲和儿子利用一个银行账户存取资金。你先运行没有互斥同步代码，捕捉账户余额产生错误的运行过程，设法截屏验证此事。然后，在用信号量解决互斥同步，保证账户操作结果正确

   应用2：用你的C库，编程解决多线程问题：父线程f创建二个子线程s和d，大儿子线程s反复向父线程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)，当二个子线程分别将一个祝福写到共享数组a和一个水果放进果盘(变量)后，父线程才去享受：从数组a收取出一个祝福和吃一个水果，如此反复进行若干次

   应用3：读者-写者问题，公平方案

五、实验方案

实验环境

1、实验运行环境：Windows10

2、虚拟机软件：VirtualBox和DOSBox

3、NASM编译器

4、TCC+TASM+TLINK混合编译

实验工具

1、编程语言：NASM，TASM，C

2、文本编辑器：Notepad++

3、软盘编辑软件：WinHex

实验思想

信号量实现

1）信号量：一个整数和一个指针组成的结构体，在内核可以定义若干个信号量，统一编号。

2）内核实现do_p()原语，在c语言中用p(int sem_id)调用

3）内核实现do_v()原语，在c语言中用v(int sem_id)调用

4）内核实现do_getsem()原语，在c语言中用getsem(int)调用,参数为信号量的初值

5）内核实现do_freesem(int sem_id),在c语言中用freesem(int sem_id)调用

信号量机制

1）信号量是内核实现的。每个信号量是一个结构体，包含两个数据域：数值count和阻塞队列头指针next。我们在内核定义一个信号量数组，同时，内核还要实现p操作和v操作。

2）我们设置4个系统调用为用户使用信号量机制。

3）int getSem(int value): 向内核申请一个内核可用信号量，并将其count域初始化为value，返值就是内核分配的一个可用信号量在数组的下标，如果没有可用的信号量，返值为 - 1。

模块结构

柱面号	磁头号	扇区号	扇区数（大小）	内容
0	0	1	1（512 B）	引导程序
0	0	2	1（512 B）	用户程序1
0	0	3	1（512 B）	用户程序2
0	0	4	1（512 B）	用户程序3
0	0	5	1（512 B）	用户程序4
0	0	6~7	2（897 B）	混合编译用户程序：输入输出
0	0	8~9	2（959 B）	多线程用户程序：统计字符个数
0	1	1~3	3（1072 B）	父子存款取款（无互斥）
0	1	4~6	3（1136 B）	父子存款取款（有互斥）
0	1	7~9	3（1280 B）	祝福与水果问题
0	1	10~12	3（1472 B）	读者-写者问题
1	0	1~11	11（5200 B）	操作系统内核

命令功能

命令	功能
help	显示支持的命令及其功能
cls	清屏
run-	并发运行任意数量的用户程序，如“run-1234"
int22h	显示“INT22H”
test	运行混合编译用户程序（输入输出）
count	运行多线程用户程序（统计字符个数）
p1	父子存款取款（无互斥）
p2	父子存款取款（有互斥）
p3	祝福与水果问题
p4	读者-写者问题
ls	显示文件列表
reboot	重启
poweroff	退出操作系统并关机

六、实验过程

线程控制过程

fork()

fork改进为可以fork出多个线程，-1代表派生失败，0代表父进程，1、2、3为子进程的编号

```
void do_fork() {
    int i = (curPCB->pid + 1) % MaxNumOfProcess;
    int num = curPCB->cpu.bx;/*派生的线程数*/
    while (i != curPCB->pid) {
        if (pcbList[i].state == NEW) {
            memcpy(&pcbList[curPCB->pid], &pcbList[i], sizeof(PCB));
            memcpy(StackList[curPCB->pid], StackList[i], StackSize);
            pcbList[i].state = READY;
            pcbList[i].fid = curPCB->pid;
            pcbList[i].pid = i;
            pcbList[i].cpu.sp += ((i - curPCB->pid) * StackSize);
            pcbList[i].cpu.ax = num;/*子进程的编号*/
            NumOfProcess++;
            num--;
            if (num == 0) {
                curPCB->cpu.ax = 0;/*父进程*/
                return;
            }
        }
        i = (i + 1) % MaxNumOfProcess;
```

```
    }
    curPCB->cpu.ax = -1;/*创建失败*/
}
```

封装为系统调用，参数为派生的线程数

```
public _fork
_fork proc
    push bp
    mov bp,sp
    push bx
    mov bx,[bp+4];
    mov ah,5h
    int 21h
    pop bx
    pop bp
    ret
_fork endp
```

用5h号系统调用实现

```
fork0:
    call _save
    call near ptr _do_fork
    jmp _restart
```

exit()

当父进程阻塞时唤醒父进程，否则直接退出

```
void do_exit() {
    if (pcbList[curPCB->fid].state = BLOCKED)
        pcbList[curPCB->fid].state = READY;
    curPCB->state = NEW;
    NumOfProcess--;
    schedule();
}
```

封装为系统调用

```
public _exit
_exit proc
    mov ah,6h
    int 21h
    ret
_exit endp
```

用6h号系统调用实现

```
exit0:
    call _save
    call near ptr _do_exit
    jmp _restart
```

wait()

wait改进为父进程在检查子进程未返回时设为阻塞态，否则就会无法唤醒。并且多个子进程结束会有多次唤醒，需要阻塞自身多次。

```
void do_wait() {
    int i;
    for (i = 0; i < MaxNumOfProcess; ++i) {
        if (pcbList[i].state != NEW && pcbList[i].fid == curPCB->pid && i != 
curPCB->pid) { /*遍历所有子进程*/
            curPCB->state = BLOCKED;
            curPCB->cpu.ax = 0;
            schedule();
            return;
        }
    }
    curPCB->cpu.ax = -1;/*已无子进程，返回-1*/
}
```

封装为系统调用

```
public _wait
_wait proc
    mov ah,7h
    int 21h
    ret
_wait endp
```

用7h号系统调用实现

```
wait0:
    call _save
    call near ptr _do_wait
    jmp _restart
```

信号量机制

信号量框架

定义信号量结构体，创建信号量队列

```
typedef struct {/*信号量*/
    int count;/*信号量的值*/
    PCB *next;/*阻塞队列*/
    char used;/*0表示未占用，1表示占用*/
}semaphone;

semaphone semList[10];/*信号量队列*/
```

初始化信号量队列，全部未占用

```
void initSem() {
    int i;
    for (i = 0; i < NRsem; i++) {
        semList[i].used = 0;
        semList[i].next = 0;
    }
}
```

GetSema的实现

GetSema()用于获取一个信号量，参数为信号量的初始值

```
void do_GetSema() {
    int i;
    for (i = 0; i < NRsem; i++) {
        if (semList[i].used == 0) {
            semList[i].used = 1;
            semList[i].count = curPCB->cpu.bx;/*获取信号量初始值*/
            curPCB->cpu.ax = i;/*返回信号量编号*/
            return;
        }
    }
    curPCB->cpu.ax = -1;/*获取信号量失败*/
}
```

封装为系统调用，参数为信号量初始值

```
public _GetSema
_GetSema proc
    push bp
    mov bp,sp
    push bx
    mov bx,[bp+4]
    mov ah,8h
    int 21h
    pop bx
    pop bp
    ret
_GetSema endp
```

用8h号系统调用实现

```
do_GetSema0:
    call _save
    call _do_GetSema
    jmp _restart
```

FreeSema的实现

FreeSema()用于释放一个信号量，注意调用前要确保所有子进程已结束

```
void do_FreeSema() {
    semList[curPCB->cpu.bx].used = 0;
}
```

封装为系统调用，参数为要释放的信号量

```
public _GetSema
_GetSema proc
    push bp
    mov bp,sp
    push bx
    mov bx,[bp+4]
    mov ah,8h
    int 21h
    pop bx
    pop bp
    ret
_GetSema endp
```

用9h号系统调用实现

```
do_FreeSema0:
    call _save
    call _do_FreeSema
    jmp _restart
```

P操作的实现

对一个信号量进行P操作，如果信号量的值小于0，则阻塞该进程

```
void do_P() {
    semList[curPCB->cpu.bx].count--;/*信号量的值减1*/
    if (semList[curPCB->cpu.bx].count < 0) {
        curPCB->state = BLOCKED;/*阻塞当前进程*/
        curPCB->next = semList[curPCB->cpu.bx].next;/*加入阻塞队列*/
        semList[curPCB->cpu.bx].next = curPCB;
        schedule();/*调度进程*/
    }
}
```

封装为系统调用，参数为进行P操作的信号量

```
public _P
_P proc
    push bp
    mov bp,sp
    push bx
    mov bx,[bp+4]
    mov ah,0Ah
    int 21h
    pop bx
    pop bp
    ret
_P endp

用0Ah号系统调用实现

~~~assembly
do_P0:
```

```
    call _save
    call _do_P
    jmp _restart
```

V操作的实现

对一个信号量进行V操作，如果信号量的值小于等于0，则从阻塞队列唤醒一个进程

```
void do_V() {
    PCB *temp;
    semList[curPCB->cpu.bx].count++;/*信号量的值加1*/
    if (semList[curPCB->cpu.bx].count <= 0) {
        temp = semList[curPCB->cpu.bx].next;
        semList[curPCB->cpu.bx].next = semList[curPCB->cpu.bx].next->next;
        temp->state = READY;/*从阻塞队列唤醒一个进程*/
    }
}
```

封装为系统调用，参数为进行V操作的信号量

```
public _V
_V proc
    push bp
    mov bp,sp
    push bx
    mov bx,[bp+4]
    mov ah,0Bh
    int 21h
    pop bx
    pop bp
    ret
_V endp
```

用0Bh号系统调用实现

```
do_V0:
    call _save
    call _do_V
    jmp _restart
```

多线程同步用户程序

父子存款取款（无互斥）

父亲和儿子利用一个银行账号存取资金。父亲存钱，每次存10元；儿子取钱，每次取20元；账号初始余额为1000元。 无互斥保护，账户余额会发生错误。

```
#include "clibs.c"

extern int fork();
extern void exit();
extern void wait();
extern void printf();

void delay() {
```

```
    int i = 0;
    int j = 0;
    while (i < 10000) {
        i++;
        j = 0;
        while (j < 10000)
            j++;
    }
}

int main() {
    int pid, i, t;
    int totalSave = 0, totalDraw = 0;
    int bankBalance = 1000;
    pid = fork(1);
    if (pid == -1) {
        printf("error in fork\r\n");
        return;
    }
    if (pid == 0) {/*父进程存钱，每次10元*/
        for (i = 0; i < 5; i++) {
            t = bankBalance;
            delay();
            t += 10;
            delay();
            bankBalance = t;
            totalSave += 10;
            printf("bankBalance=%d,totalSave=%d\r\n", bankBalance, totalSave);
        }
        wait();
    }
    else if (pid == 1) {/*子进程取钱，每次20元*/
        for (i = 0; i < 5; i++) {
            t = bankBalance;
            delay();
            t -= 20;
            delay();
            bankBalance = t;
            totalDraw += 20;
            printf("bankBalance=%d,totalDraw=%d\r\n", bankBalance, totalDraw);
        }
        exit();
    }
}
```

父子存款取款（有互斥）

父亲和儿子利用一个银行账号存取资金。父亲存钱，每次存10元；儿子取钱，每次取20元；账号初始余额为1000元。 有互斥保护，利用信号量实现临界区，在临界区中只有一个进程可以执行。

```
#include "clibs.c"

extern int fork();
extern void wait();
extern void exit();
extern void printf();
extern int GetSema();
```

```
extern void FreeSema();
extern void P();
extern void V();

void delay() {
    int i = 0;
    int j = 0;
    while (i < 10000) {
        i++;
        j = 0;
        while (j < 10000)
            j++;
    }
}

int main() {
    int pid, i, t, s;
    int totalSave = 0, totalDraw = 0;
    int bankBalance = 1000;
    s = GetSema(1);
    pid = fork(1);
    if (pid == -1) {
        printf("error in fork\r\n");
        return;
    }
    if (pid == 0) {/*父进程存钱，每次10元*/
        for (i = 0; i < 5; i++) {
            P(s);
            t = bankBalance;
            delay();
            t += 10;
            delay();
            bankBalance = t;
            totalSave += 10;
            printf("bankBalance=%d,totalSave=%d\r\n", bankBalance, totalSave);
            V(s);
        }
        wait();
        FreeSema(s);
    }
    else if (pid == 1) {/*子进程取钱，每次20元*/
        for (i = 0; i < 5; i++) {
            P(s);
            t = bankBalance;
            delay();
            t -= 20;
            delay();
            bankBalance = t;
            totalDraw += 20;
            printf("bankBalance=%d,totalDraw=%d\r\n", bankBalance, totalDraw);
            V(s);
        }
        exit();
    }
}
```

祝福与水果问题

父进程创建两个子线程，“大儿子”送祝福，“小儿子”送水果，只有祝福和水果都齐了“父亲”才能享受。主要要对祝福和水果都分别设置信号量。

```
#include "clibs.c"

extern int fork();
extern void wait();
extern void exit();
extern void printf();
extern int GetSema();
extern void FreeSema();
extern void P();
extern void V();

void strcpy(char *a, char *b) {
    int i;
    for (i = 0; b[i] != 0; i++)
        a[i] = b[i];
    a[i] = 0;
}

char words[80];/*祝福，word[0]==0时表示无祝福*/

int main() {
    int pid, i;
    int fs, ws;/*两个信号量，代表水果与祝福*/
    int fruit = 0;/*0表示空，1表示苹果，2表示香蕉*/
    words[0] = 0;
    fs = GetSema(1);
    ws = GetSema(1);
    pid = fork(2);
    if (pid == -1) {
        printf("error in fork\r\n");
        return;
    }
    if (pid == 0) {/*父亲*/
        for (i = 0; i < 5; i++) {
            P(ws);
            P(fs);
            if (words[0] == 0 || fruit == 0)
                i--;
            else {/*祝福和水果都送齐了父亲才能享受*/
                if (fruit == 1)
                    printf("words: %s fruit: apple\r\n", words);
                else
                    printf("words: %s fruit: banana\r\n", words);
                words[0] = 0;/*取走祝福和水果*/
                fruit = 0;
            }
            V(fs);
            V(ws);
        }
        wait();
        FreeSema(fs);
        FreeSema(ws);
```

```
    }
    else if (pid == 1) {/*大儿子送祝福*/
        for (i = 0; i < 5; i++) {
            P(ws);
            if (words[0] != 0)
                i--;
            else
                strcpy(words, "Father will live one year after anther for
ever!");
            V(ws);
        }
        exit();
    }
    else if (pid == 2) {/*小儿子送水果*/
        for (i = 0; i < 5; i++) {
            P(fs);
            if (fruit != 0)
                i--;
            else
                fruit = i % 2 + 1;
            V(fs);
        }
        exit();
    }
}
```

读者写者问题（公平策略）

采用公平策略，读者和写者有相同的优先权。读者和写者按照先后顺序对数据进行读和写。因此，一共需要3个信号量。一个用于互斥写者和读者，一个用于互斥写者和写者，同时由于要记录正在读的读者的数量，所以还需要一个信号量用于读者和读者，才能正确更新正在读的读者数量值。

```
#include "clibs.c"

extern int fork();
extern void wait();
extern void exit();
extern void printf();
extern int GetSema();
extern void FreeSema();
extern void P();
extern void V();

int ReaderNum = 3;/*3个读者*/
int WriterNum = 2;/*2个写者*/
int reader_counter = 0;/*正在读的读者*/

int main() {
    int wmutex, rmutex, mutex;/*互斥量：写者、读者、读者和写者*/
    int pid, i;
    wmutex = GetSema(1);
    rmutex = GetSema(1);
    mutex = GetSema(1);
    pid = fork(ReaderNum + WriterNum);
    if (pid == -1) {
        printf("error in fork\r\n");
        return;
```

```
    }
    if (pid == 0) {/*父进程*/
        wait();
        FreeSema(wmutex);
        FreeSema(rmutex);
        FreeSema(mutex);
    }
    else if (pid >= 1 && pid <= ReaderNum) {/*读者*/
        for (i = 0; i < 2; i++) {
            P(rmutex);
            if (reader_counter == 0)
                P(mutex);/*写者写完，读者才能读*/
            reader_counter++;
            V(rmutex);
            printf("reader %d is reading, num of readers %d\r\n", pid - 1,
reader_counter);
            P(rmutex);
            reader_counter--;
            if (reader_counter == 0)
                V(mutex);/*所有进入的读者都读完了，写者才能写*/
            V(rmutex);
        }
        exit();
    }
    else {/*写者*/
        for (i = 0; i < 3; i++) {
            P(wmutex);/*只有一个写者可以进入*/
            P(mutex);/*所有读者读完，才能写*/
            printf("writer %d is writing\r\n", pid - ReaderNum - 1);
            V(mutex);
            V(wmutex);
        }
        exit();
    }
}
```

内核其他函数修改

因为这次实验我的用户程序一共超过了18个扇区，无法在一个磁头的扇区内，所以我改进了我跳转用户程序的汇编函数，新增了一个磁头号作为参数。这样，新的loadProgram()共有5个参数，cs段地址，ip偏移地址，head磁头号，id起始扇区号，num扇区数，就能实现跳转到柱面号为0的任意扇区（前36个扇区）。

```
; 加载用户程序5个参数(cs段地址，ip偏移地址，head磁头号，id起始扇区号，num扇区数)
public _loadProgram
_loadProgram proc
    push bp
    mov bp,sp

    push ax
    push bx
    push cx
    push dx
    push es

    mov ax,[bp+4]            ; 段地址，存放数据的内存基地址
```

```
    mov es,ax                   ; 设置段地址（不能直接mov es,段地址）
    mov bx,[bp+6]               ; 偏移地址; 存放数据的内存偏移地址
    mov ah,2                    ; 功能号2
    mov al,[bp+12]              ; 扇区数
    mov dl,0                    ; 驱动器号 ; 软盘为0，硬盘和U盘为80H
    mov dh,[bp+8]               ; 磁头号，起始编号为0
    mov ch,0                    ; 柱面号，起始编号为0
    mov cl,[bp+10]              ; 起始扇区号 ; 起始编号为1
    int 13H                     ; 调用读磁盘BIOS的13h功能
    ;用户程序已加载到指定内存区域

    pop es
    pop dx
    pop cx
    pop bx
    pop ax
    pop bp
    ret
_loadProgram endp
```

相应地，C中的creatProcess()创建进程函数也要增加磁头号head参数

```
int creatProcess(int head, int id, int num) {
    int i, seg;
    for (i = 0; i < MaxNumOfProcess; i++)
        if (pcbList[i].state == NEW)
            break;
    if (i == MaxNumOfProcess) {
        printf("ERROR: Fail to creat process!\r\n");
        return 0;
    }
    seg = (i + 2) * 0x1000;
    pcbList[i].cpu.cs = seg;
    pcbList[i].cpu.ds = seg;
    pcbList[i].cpu.es = seg;
    pcbList[i].cpu.ip = 0x100;
    pcbList[i].cpu.ss = 0x1000;
    pcbList[i].cpu.sp = InitStack(&StackList[i][StackSize]);/*初始化进程栈*/
    pcbList[i].cpu.flags = 512;
    pcbList[i].pid = i;
    pcbList[i].fid = i;
    pcbList[i].name[0] = '0' + i;
    pcbList[i].name[1] = 0;
    loadReady(seg);
    loadProgram(seg, 0x100, head, id, num);/*修改为5个参数的loadProgram汇编函数*/
    pcbList[i].state = READY;
    NumOfProcess += 1;
    return i;
}
```

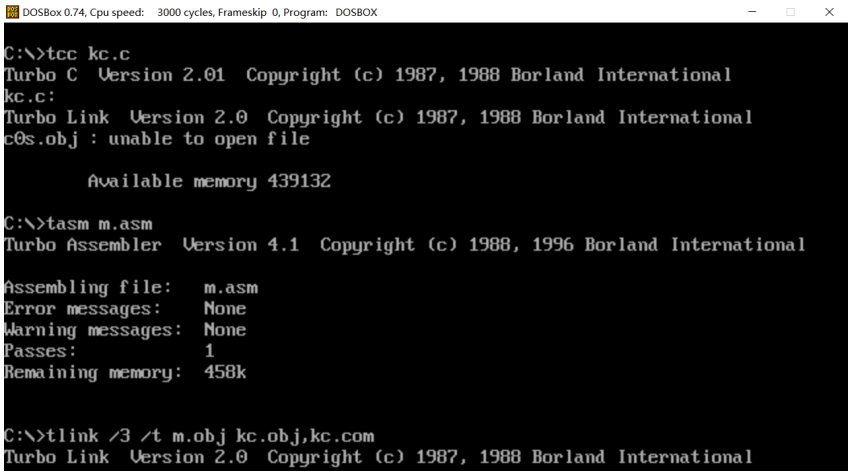
完整的用于执行所有命令的函数如下：

```
void RunCom(char *command) {
    int i = 4;
    if(!strcmp(command,"help")) {
        help();
```

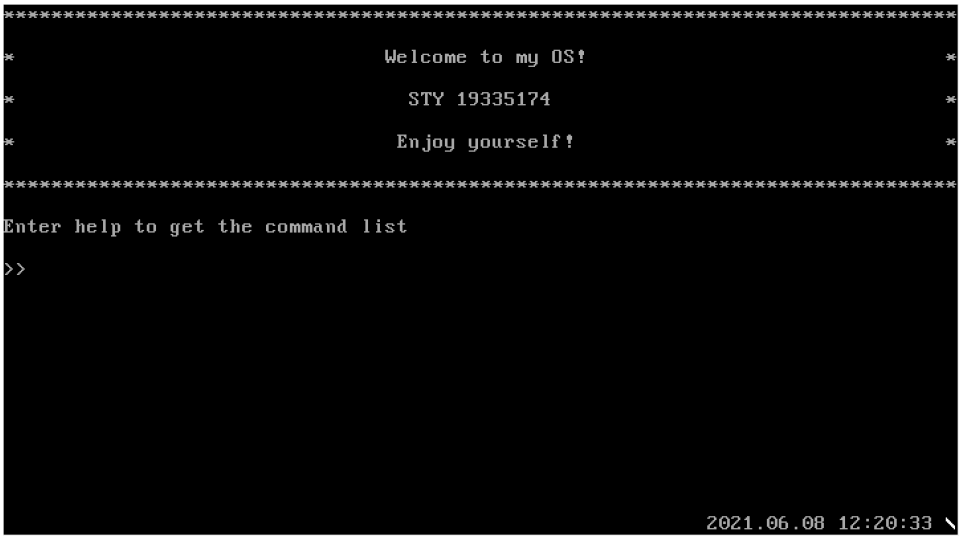
```
    }
    else if(!strcmp(command,"cls")) {
        cls();
    }
    else if(!strcmp(command,"ls")) {
        file();
    }
    else if(command[0]=='r'&&command[1]=='u'&&command[2]=='n'&&command[3]=='-')
{
        while (command[i] == '1' || command[i] == '2' || command[i] == '3' ||
command[i] == '4') {
            if (command[i] == '1') creatProcess(0, 2, 1);
            if (command[i] == '2') creatProcess(0, 3, 1);
            if (command[i] == '3') creatProcess(0, 4, 1);
            if (command[i] == '4') creatProcess(0, 5, 1);
            i++;
        }
    }
    else if(!strcmp(command,"test")) {
        creatProcess(0, 6, 2);
        cls();
    }
    else if(!strcmp(command,"count")) {
        creatProcess(0, 8, 2);
        cls();
    }
    else if(!strcmp(command,"p1")) {
        creatProcess(1, 1, 3);
        cls();
    }
    else if(!strcmp(command,"p2")) {
        creatProcess(1, 4, 3);
        cls();
    }
    else if(!strcmp(command,"p3")) {
        creatProcess(1, 7, 3);
        cls();
    }
    else if(!strcmp(command,"p4")) {
        creatProcess(1, 10, 3);
        cls();
    }
    else if(!strcmp(command,"int22h")) {
        int22h();
    }
    else if(!strcmp(command,"reboot")) {
        Reboot();
    }
    else if(!strcmp(command,"poweroff")) {
        symbol = 0;
    }
    else {
        printf("\n\rWrong command!\n\r");
    }
    while (NumOfProcess != 0);/*阻塞，直至进程都完成*/
}
```

七、实验结果

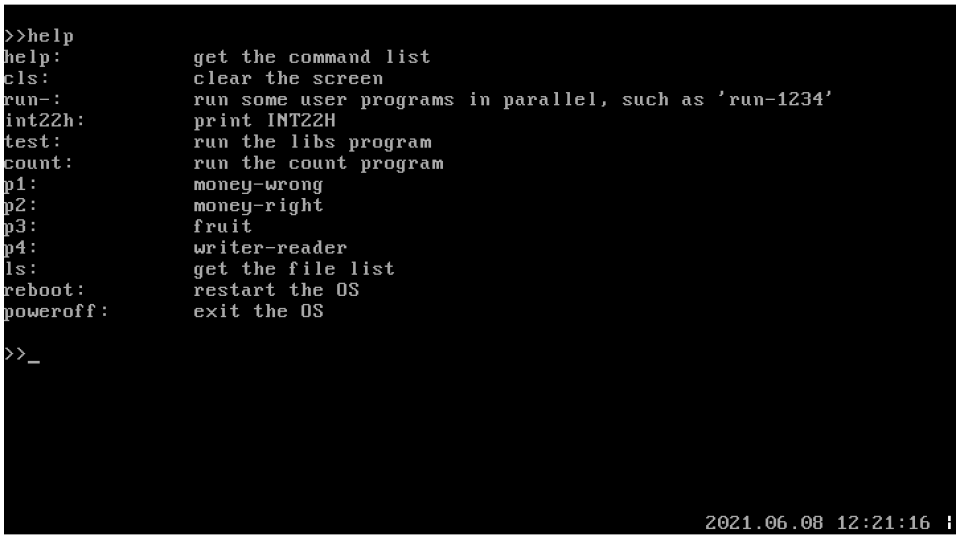
内核混合编译



操作系统界面



帮助菜单



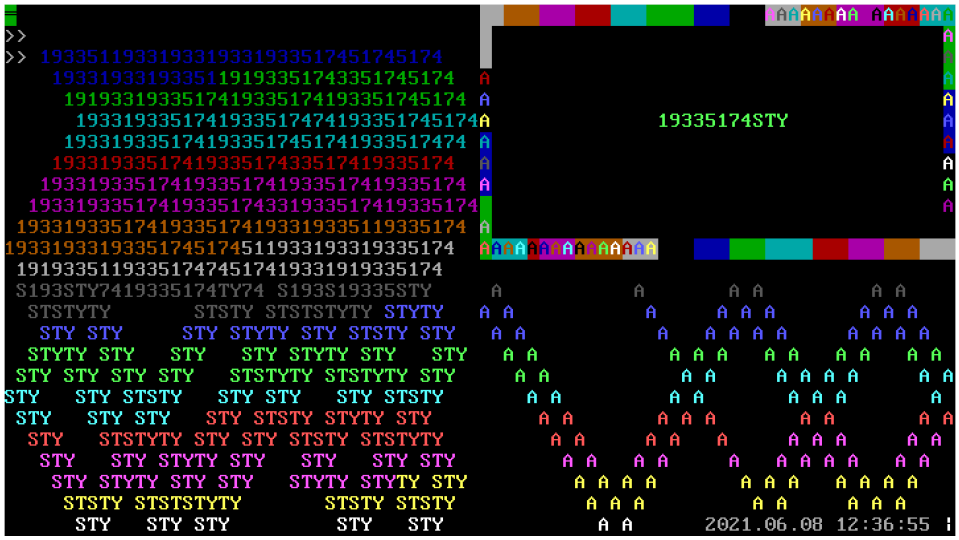
文件列表

```
>>ls
The Number Of Files: 10
01.Number           402B
02.Name             397B
03.Rectangle        298B
04.Stone            393B
05.Test             897B
06.Count            959B
07.money-wrong      1086B
08.money-right      1148B
09.fruit            1295B
10.writer-reader    1475B

>>_
```

2021.06.08 12:35:40

4个用户程序并行



父子存款取款（无互斥）

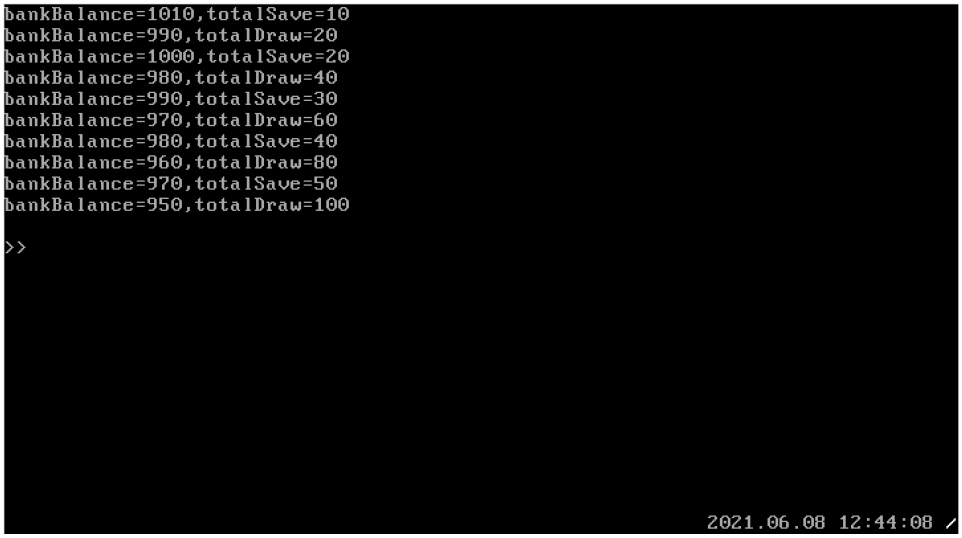
```
bankBalance=1010,totalSave=10
bankBalance=980,totalDraw=20
bankBalance=1020,totalSave=20
bankBalance=960,totalDraw=40
bankBalance=1030,totalSave=30
bankBalance=940,totalDraw=60
bankBalance=1040,totalSave=40
bankBalance=920,totalDraw=80
bankBalance=1050,totalSave=50
bankBalance=900,totalDraw=100

>>
```

2021.06.08 12:42:14

结果错误，无互斥方案和有互斥方案的余额完全不同！

父子存款取款（有互斥）

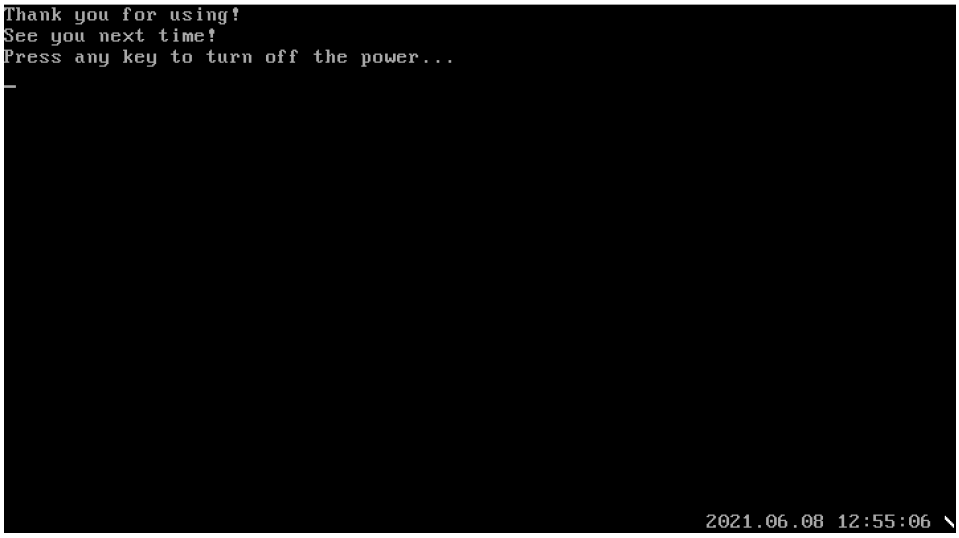


祝福与水果问题

读者-写者问题

关闭电源（int 21h的3号功能）

按下任意按键后关闭电源！



八、实验总结

这次实验新增了信号量的机制，也新增了4个使用信号量的混合编译用户程序，任务量还是非常大的，也让我没日没夜地连写了好几天才“肝”完。

信号量的知识我们在理论课上已经学过，而且老师已经提供了信号量的结构体示例，还有GetSema函数，FreeSema函数，P操作和V操作的代码，所以整个信号量机制的实现并不是非常困难，只要将它们改为系统调用就能成功实现了。

我在实验七的线程函数还不够完善，因为实验八的要求更高了，派生出的线程可能不止一个，所以我不得不修改了我之前写的线程函数。还有我发现实验七的exit其实是有错的，上次调用了int 20h返回，但是其实应该改变一下父进程的状态，再用schedule调度进程。

在写“祝福与水果问题”时，我因为一开始没有理解这个问题是什么意思，以为是只要“祝福”和“水果”不是同时送就行了，导致我一开始做错了。后来经过经过同学的点拨才理解这个问题的含义，最后修改代码也成功写完了。读者写者问题我还不能完全掌握运用信号量的每个细节，所以我也是参考了书上的代码才写完，而且读者写者问题不知道具体的实现结果是怎样的，所以也疑惑了我好久，不过最终还是想了个简单的输出办法实现了。

实验过程中一个小bug困扰了我很久，我在一处代码上用while循环可以成功执行，而用for就不行。后来才发现因为我那个循环写的时for (int i...)，而tcc要求局部变量只能在函数开头声明，所以要把循环变量i写在函数的开头，这个无语的bug浪费了我很多时间。还有因为这次实验新增的代码比较多，导致我的内核占了11个扇区，新加的4个信号量用户程序也一共占了12个扇区，所以我原来的汇编跳转扇区函数失效了。不过我将4个信号量用户程序放在柱面号0磁头号1的扇区部分，将内核放在柱面号1磁头号0的扇区部分，给汇编函数新加了一个磁头号的参数，就能成功解决了。

总而言之，这次实验难度比较大，要写的代码很多，要解决的问题也很多，所以还是花费了我大量的时间。这次实验也加深了我对信号量和互斥问题的理解，自己对于互斥问题也能成功解决，还是学到了很多东西的。

操作系统的实验已经接近尾声，回顾整个学期，虽然每一次的实验都让我心力交瘁，但也是在一次次实验中才让我蜕变成长。感谢老师和助教们的付出，让我受益匪浅。“路漫漫其修远兮，吾将上下而求索”，或许后面的操作系统实验我已经没有精力继续探究了，但操作系统实验教会我不断钻研的精神，将永远伴随吾身！

九、参考文献

《汇编语言 （第3版）》

《X86汇编语言：从实模式到保护模式》

https://wenku.baidu.com/view/9ac0446c48d7c1c708a145aa.html

https://blog.csdn.net/qq_35235032/article/details/106652964?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param