实验六：实现时间片轮转的二态进程模型

(15分)

实验目的：

1、学习多道程序与CPU分时技术

2、掌握操作系统内核的二态进程模型设计与实现方法

3、掌握进程表示方法

4、掌握时间片轮转调度的实现

实验要求：

1、了解操作系统内核的二态进程模型

2、扩展实验五的的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。

4、修改时钟中断处理程序，调用时间片轮转调度算法。

5、设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。

5、修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场。

6、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

实验内容：

1. 修改实验5的内核代码，定义进程控制块PCB类型，包括进程号、程序名、进程内存地址信息、CPU寄存器保存区、进程状态等必要数据项，再定义一个PCB数组，最大进程数为10个。

define MaxProcessNo=10

typedef strut {

int ax;

int bx;

int cx;

int dx;

int cs;

int ds;

int es;

int ss;

int sp;

int bp;

int di;

int si;

int ip;

int flag;

} cpuRegisters

typedef struct {

cpuRegisters cpuRegs;

int pid;

char pname[10];

char pstate;

…

} PCB;

PCB pcblist[MaxProcessNo]

1. 扩展实验五的的内核程序，增加一条命令可同时执行多个用户程序，内核加载这些程序，创建多个进程，再实现分时运行
2. 修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程

Timer:

save()

call showWingFireWheel() ；无敌风火轮显示

call \_schedule() ; 调用进程调度过程

jmp restart

1. 内核增加进程调度过程：每次调度，将当前进程转入就绪状态，选择下一个进程运行，如此反复轮流运行。

void schedule(){

CurrentProcessNo++；

if (CurrentProcessNo=MaxProcessNo)

CurrentProcessNo=0；

}

1. 修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的运行。

;参考程序

;Minix的save和restart

; save

;=====================================================

save: ; save the machine state in the proc table.

push ds ; stack: psw/cs/pc/ret addr/ds

push cs ; prepare to restore ds

pop ds ; ds has now been set to cs

mov ds,ker\_ds ; word 4 in kernel text space contains ds value

pop ds\_save ; stack: psw/cs/pc/ret addr

pop ret\_save ; stack: psw/cs/pc

mov bx\_save,bx ; save bx for later ; we need a free register

mov bx,dgroup:proc\_ptr ; start save set up; make bx point to save area

add bx,OFF ; bx points to place to store cs

pop PC-OFF[bx] ; store pc in proc table

pop csreg-OFF[bx] ; store cs in proc table

pop PSW-OFF[bx] ; store psw

mov ssreg-OFF[bx],ss ; store ss

mov spreg-OFF[bx],sp ; sp as it was prior to interrupt

mov sp,bx ; now use sp to point into proc table/task save

mov bx,ds ; about to set ss

mov ss,bx ; set ss

push ds\_save ; start saving all the registers, sp first

push es ; save es between sp and bp

mov es,bx ; es now references kernel memory too

push bp ; save bp

push di ; save di

push si ; save si

push dx ; save dx

push cx ; save cx

push bx\_save ; save original bx

push ax ; all registers now saved

mov sp,offset dgroup:k\_stack ; temporary stack for interrupts

add sp,K\_STACK\_BYTES ; set sp to top of temporary stack

mov splimit,offset dgroup:k\_stack ; limit for temporary stack

add splimit,8 ; splimit checks for stack overflow

mov ax,ret\_save ; ax = address to return to

jmp ax ; return to caller; Note: sp points to saved ax

;==================================================

; restart

;==================================================

restart: ; This routine sets up and runs a proc or task.

cmp dgroup:cur\_proc,IDLE; restart user; if cur\_proc = IDLE, go idle

je \_idle ; no user is runnable, jump to idle routine

cli ; disable interrupts

mov sp,dgroup:proc\_ptr ; return to user, fetch regs from proc table

pop ax ; start restoring registers

pop bx ; restore bx

pop cx ; restore cx

pop dx ; restore dx

pop si ; restore si

pop di ; restore di

mov lds\_low,bx ; lds\_low contains bx

mov bx,sp ; bx points to saved bp register

mov bp,SPLIM-ROFF[bx] ; splimit = p\_splimit

mov splimit,bp ; ditto

mov bp,dsreg-ROFF[bx] ; bp = ds

mov lds\_low+2,bp ; lds\_low+2 contains ds

pop bp ; restore bp

pop es ; restore es

mov sp,spreg-ROFF[bx] ; restore sp

mov ss,ssreg-ROFF[bx] ; restore ss using the value of ds

push PSW-ROFF[bx] ; push psw (flags)

push csreg-ROFF[bx] ; push cs

push PC-ROFF[bx] ; push pc

lds bx,DWORD PTR lds\_low ; restore ds and bx in one fell swoop

iret ; return to user or task

1. 实验5的内核其他功能，如果不必要，可暂时取消服务。

用户程序d

用户程序c

用户程序a

用户程序b

监控程序(内核)

引导程序

**实验环境：**

⚫ Windows 10-64bit

⚫ Vmware WorkStation 15 pro 15.5.1 build-15018445：虚拟机软件

⚫ NASM version 2.13.02：汇编程序的编译器，在 linux 下通过

sudo apt-get install nasm 下载

⚫ Ubuntu-18.04.4:安装在 Vmware 的虚拟机上

⚫ 代码编辑器：Visual Studio Code 1.44.2

⚫ GNU ld 2.30：链接器

**实验思路：**

本次实验的主要目的是实现多进程模型，因此将以往的系统中一些部分功能进行了调整（如run的功能，这个之后会详细介绍），同时对以前实现过的功能的介绍会变得简单一些。

同样地，先将操作系统的结构及其功能的划分通过表格的方式列出来，如下表所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 柱面号 | 磁头号 | 扇区号 | 扇区大小 | 功能 |
| 0 | 0 | 1 | 1个扇区 | 引导程序 |
| 0 | 0 | 2 | 1个扇区 | 用户信息表 |
| 0 | 0到1 | 0号柱面3号扇区到1号柱面18号扇区 | 34个扇区 | 内核 |
| 1 | 0 | 1到2 | 2个扇区 | 用户程序b |
| 1 | 0 | 3到4 | 2个扇区 | 用户程序a |
| 1 | 0 | 5到6 | 2个扇区 | 用户程序c |
| 1 | 0 | 7到8 | 2个扇区 | 用户程序d |
| 1 | 0 | 9 | 1个扇区 | 实验4调用int33h~36h的用户程序 |
| 1 | 0 | 10 | 3个扇区 | 展示系统调用的用户程序 |

在上表中，引导程序和用户信息表与往常无异，在此就不再赘述了。

具体有较大变动的是内核部分，下面是对内核部分的表格简述。

|  |  |
| --- | --- |
| 文件名 | 功能 |
| osstarter.asm | 监控程序，接收用户命令，执行相应的用户程序 |
| liba.asm | 包含n个汇编编写的函数 |
| kernel.c | 包含n个C编写的函数 |
| stringio.h | kernel.c的头文件，实现了输入输出等功能 |
| systema.asm | 包含n个汇编编写的系统调用函数 |
| systemc.c | 包含n个汇编编写的系统调用函数的辅助函数 |
| multiprocess.asm | 多线程模型，实现了定义进程控制块PCB类型、寄存器的保护与恢复等功能。 |

在本次实验的操作系统内核中，将以往的一些命令给删除了（如风火轮(hotwheel)），保留了大部分命令（如help、clear等），修改了run命令，并且把原来的run命令的功能（即多道批处理）移植到了新增的命令bat上。而run命令的功能则变成创建多个线程并发执行用户程序。

由于本次实验中的用户程序可以被并发执行，且与内核处在同一个段之中，故将原用户程序的地址进行了修改，以方便进程切换。具体修改如下：

原地址：

offset\_userprog1 equ 0A300h

offset\_userprog2 equ 0A700h

offset\_userprog3 equ 0AB00h

offset\_userprog4 equ 0AF00h

offset\_intcaller equ 0xB300

修改后的地址：

addr\_userprog1 equ 10000h

addr\_userprog2 equ 20000h

addr\_userprog3 equ 30000h

addr\_userprog4 equ 40000h

同样被修改的地址还有intcaller，在此就不做赘述了。

**下面介绍本次实验的着重部分：multiprocess.asm**

multiprocess.asm主要实现了三大功能：寄存器保护→进程的调度→寄存器恢复。由于只有多线程运行用户程序时操作系统才执行上述的操作，所以我们可以设置一个变量flag=0，当用户执行run命令时，将flag设置为1，此时表示将要运行多线程，并执行相应的操作。当执行完毕之后，再将flag给设置为0。关键代码如下：

Timer:                             ; 08h号时钟中断处理程序

    cmp word[cs:timer\_flag], 0

je QuitTimer

;寄存器保护代码

;进程调度代码

;寄存器恢复代码

QuitTimer:

    push ax

    mov al, 20h

    out 20h, al

    out 0A0h, al

    pop ax

    iret

    timer\_flag dw 0

    current\_process\_id dw 0

以上就是设置了标志变量之后时钟中断程序的模板，下面叙述其中的三大功能的代码：

**寄存器保护：**

具体思路是：在flag=1之后，将所有寄存器的值压栈，然后调用我们写好的pcbSave函数，该函数的作用是，从栈上取得寄存器的值并将其保存在当前进程的PCB中。其核心代码如下：

pcbSave:                           ; 函数：现场保护

    pusha

    mov bp, sp

    add bp, 16+2                   ; 参数首地址

    mov di, pcb\_table

    mov ax, 34

    mul word[cs:current\_process\_id]

    add di, ax                     ; di指向当前PCB的首地址

    mov ax, [bp]

    mov [cs:di], ax

    mov ax, [bp+2]

    mov [cs:di+2], ax

    mov ax, [bp+4]

    mov [cs:di+4], ax

    mov ax, [bp+6]

    mov [cs:di+6], ax

    mov ax, [bp+8]

    mov [cs:di+8], ax

    mov ax, [bp+10]

    mov [cs:di+10], ax

    mov ax, [bp+12]

    mov [cs:di+12], ax

    mov ax, [bp+14]

    mov [cs:di+14], ax

    mov ax, [bp+16]

    mov [cs:di+16], ax

    mov ax, [bp+18]

    mov [cs:di+18], ax

    mov ax, [bp+20]

    mov [cs:di+20], ax

    mov ax, [bp+22]

    mov [cs:di+22], ax

    mov ax, [bp+24]

    mov [cs:di+24], ax

    mov ax, [bp+26]

    mov [cs:di+26], ax

    mov ax, [bp+28]

    mov [cs:di+28], ax

    mov ax, [bp+30]

    mov [cs:di+30], ax

    popa

    ret

在这里我们使用了进程控制块（PCB）和进程表的内容，在此定义了8个进程控制块（0~7），下面给出相关的定义代码：

%macro ProcessControlBlock 0       ; 参数：段值

    dw 0                           ; ax，偏移量=+0

    dw 0                           ; cx，偏移量=+2

    dw 0                           ; dx，偏移量=+4

    dw 0                           ; bx，偏移量=+6

    dw 0FE00h                      ; sp，偏移量=+8

    dw 0                           ; bp，偏移量=+10

    dw 0                           ; si，偏移量=+12

    dw 0                           ; di，偏移量=+14

    dw 0                           ; ds，偏移量=+16

    dw 0                           ; es，偏移量=+18

    dw 0                           ; fs，偏移量=+20

    dw 0B800h                      ; gs，偏移量=+22

    dw 0                           ; ss，偏移量=+24

    dw 0                           ; ip，偏移量=+26

    dw 0                           ; cs，偏移量=+28

    dw 512                         ; flags，偏移量=+30

    db 0                           ; id，进程ID，偏移量=+32

    db 0                           ; state，{0:新建态; 1:就绪态; 2:运行态}，偏移量=+33

%endmacro

pcb\_table:                         ; 定义PCB表

pcb\_0: ProcessControlBlock         ; 0号PCB存放内核

pcb\_1: ProcessControlBlock

pcb\_2: ProcessControlBlock

pcb\_3: ProcessControlBlock

pcb\_4: ProcessControlBlock

pcb\_5: ProcessControlBlock

pcb\_6: ProcessControlBlock

pcb\_7: ProcessControlBlock

对于寄存器的保护过程来说，比较重要的是将寄存器压栈并将其取出来保存到当前进程这个过程，在此采用的是PCB首地址+偏移量的算法来获得当前的进程号。

**进程调度：**

具体的实现思路为：将当前进程的运行态改成就绪态，然后递增id变量指向下一个进程，直到找到下一个就绪态进程为止，即完成了调度。同时，当我们按下esc键的时候，我们就结束掉所有的进程，并返回到内核当中。核心代码如下：

pcbSchedule:                       ; 函数：进程调度

    pusha

    mov si, pcb\_table

    mov ax, 34

    mul word[cs:current\_process\_id]

    add si, ax                     ; si指向当前PCB的首地址

    mov byte[cs:si+33], 1          ; 将当前进程设置为就绪态

    mov ah, 01h                    ; 功能号：查询键盘缓冲区但不等待

    int 16h

    jz try\_next\_pcb                ; 无键盘按下，继续

    mov ah, 0                      ; 功能号：查询键盘输入

    int 16h

    cmp al, 27                     ; 是否按下ESC

    jne try\_next\_pcb               ; 若按下ESC，回到内核

    mov word[cs:current\_process\_id], 0

    mov word[cs:timer\_flag], 0     ; 禁止时钟中断处理多进程

    call resetAllPcbExceptZero

    jmp QuitSchedule

    try\_next\_pcb:                  ; 循环地寻找下一个处于就绪态的进程

        inc word[cs:current\_process\_id]

        add si, 34                 ; si指向下一PCB的首地址

        cmp word[cs:current\_process\_id], 7

        jna pcb\_not\_exceed         ; 若id递增到8，则将其恢复为1

        mov word[cs:current\_process\_id], 1

        mov si, pcb\_table+34       ; si指向1号进程的PCB的首地址

    pcb\_not\_exceed:

        cmp byte[cs:si+33], 1      ; 判断下一进程是否处于就绪态

        jne try\_next\_pcb           ; 不是就绪态，则尝试下一个进程

        mov byte[cs:si+33], 2      ; 是就绪态，则设置为运行态。调度完毕

    QuitSchedule:

    popa

    ret

**寄存器恢复：**

具体思路是，将调度侯的新的PCB里面取出各个寄存器的值并赋给原来的寄存器。当所有的寄存器恢复完毕之后，我们就可以将新进程的cs、ip等值压栈，最后中断返回即可。寄存器恢复核心代码如下：

PcbRestart:                        ; 不是函数

    mov si, pcb\_table

    mov ax, 34

    mul word[cs:current\_process\_id]

    add si, ax                     ; si指向调度后的PCB的首地址

    mov ax, [cs:si+0]

    mov cx, [cs:si+2]

    mov dx, [cs:si+4]

    mov bx, [cs:si+6]

    mov sp, [cs:si+8]

    mov bp, [cs:si+10]

    mov di, [cs:si+14]

    mov ds, [cs:si+16]

    mov es, [cs:si+18]

    mov fs, [cs:si+20]

    mov gs, [cs:si+22]

    mov ss, [cs:si+24]

    add sp, 11\*2                   ; 恢复正确的sp

    push word[cs:si+30]            ; 新进程flags

    push word[cs:si+28]            ; 新进程cs

    push word[cs:si+26]            ; 新进程ip

    push word[cs:si+12]

    pop si                         ; 恢复si

下面是时钟中断返回：

QuitTimer:

    push ax

    mov al, 20h

    out 20h, al

    out 0A0h, al

    pop ax

    iret

    timer\_flag dw 0

    current\_process\_id dw 0

至此，我们缺少的就是将用户程序加载到内存并初始化其PCB部分的内容。具体代码如下，其对应的功能可以参考注释：

loadProcessMem:                    ; 函数：将某个用户程序加载入内存并初始化其PCB

    pusha

    mov bp, sp

    add bp, 16+4                   ; 参数地址

    LOAD\_TO\_MEM [bp+12], [bp], [bp+4], [bp+8], [bp+16], [bp+20]

    mov si, pcb\_table

    mov ax, 34

    mul word[bp+24]                ; progid\_to\_run

    add si, ax                     ; si指向新进程的PCB

    mov ax, [bp+24]                ; ax=progid\_to\_run

    mov byte[cs:si+32], al         ; id

    mov ax, [bp+16]                ; ax=用户程序的段值

    mov word[cs:si+16], ax         ; ds

    mov word[cs:si+18], ax         ; es

    mov word[cs:si+20], ax         ; fs

    mov word[cs:si+24], ax         ; ss

    mov word[cs:si+28], ax         ; cs

    mov byte[cs:si+33], 1          ; state设其状态为就绪态

    popa

    retf

同时，为了让run命令可以重复使用，我们需要在每次run命令侯，将PCB表中的PCB重置为初始状态，否则下次调用就会出错。如下：

resetAllPcbExceptZero:

    push cx

    push si

    mov cx, 7                      ; 共8个PCB

    mov si, pcb\_table+34

    loop1:

        mov word[cs:si+0], 0       ; ax

        mov word[cs:si+2], 0       ; cx

        mov word[cs:si+4], 0       ; dx

        mov word[cs:si+6], 0       ; bx

        mov word[cs:si+8], 0FE00h  ; sp

        mov word[cs:si+10], 0      ; bp

        mov word[cs:si+12], 0      ; si

        mov word[cs:si+14], 0      ; di

        mov word[cs:si+16], 0      ; ds

        mov word[cs:si+18], 0      ; es

        mov word[cs:si+20], 0      ; fs

        mov word[cs:si+22], 0B800h ; gs

        mov word[cs:si+24], 0      ; ss

        mov word[cs:si+26], 0      ; ip

        mov word[cs:si+28], 0      ; cs

        mov word[cs:si+30], 512    ; flags

        mov byte[cs:si+32], 0      ; id

        mov byte[cs:si+33], 0      ; state=新建态

        add si, 34                 ; si指向下一个PCB

        loop loop1

    pop si

    pop cx

    ret

通过loop1循环将所有PCB重置。

**混合编译链接：**

将所有文件进行联合编译链接，保存为myos.sh，并运行之，得到Condor\_OS.img。

rm -rf temp

mkdir temp

rm \*.img

nasm bootloader.asm -o ./temp/bootloader.bin

nasm userproginfo.asm -o ./temp/userproginfo.bin

cd userprog

nasm b.asm -o ../temp/b.bin

nasm a.asm -o ../temp/a.bin

nasm c.asm -o ../temp/c.bin

nasm d.asm -o ../temp/d.bin

nasm interrupt\_caller.asm -o ../temp/interrupt\_caller.bin

nasm syscall\_test.asm -o ../temp/syscall\_test.bin

cd ..

cd lib

nasm -f elf32 systema.asm -o ../temp/systema.o

gcc -fno-pie -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -shared systemc.c -o ../temp/systemc.o

cd ..

nasm -f elf32 osstarter.asm -o ./temp/osstarter.o

nasm -f elf32 liba.asm -o ./temp/liba.o

nasm -f elf32 multiprocess.asm -o ./temp/multiprocess.o

gcc -fno-pie -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -shared kernel.c -o ./temp/kernel.o

ld -m elf\_i386 -N -Ttext 0x8000 --oformat binary ./temp/osstarter.o ./temp/liba.o ./temp/kernel.o ./temp/systema.o ./temp/systemc.o ./temp/multiprocess.o -o ./temp/kernel.bin

dd if=./temp/bootloader.bin of=Condor\_OS.img bs=512 count=1 2> /dev/null

dd if=./temp/userproginfo.bin of=Condor\_OS.img bs=512 seek=1 count=1 2> /dev/null

dd if=./temp/kernel.bin of=Condor\_OS.img bs=512 seek=2 count=34 2> /dev/null

dd if=./temp/b.bin of=Condor\_OS.img bs=512 seek=36 count=2 2> /dev/null

dd if=./temp/a.bin of=Condor\_OS.img bs=512 seek=38 count=2 2> /dev/null

dd if=./temp/c.bin of=Condor\_OS.img bs=512 seek=40 count=2 2> /dev/null

dd if=./temp/d.bin of=Condor\_OS.img bs=512 seek=42 count=2 2> /dev/null

dd if=./temp/interrupt\_caller.bin of=Condor\_OS.img bs=512 seek=44 count=1 2> /dev/null

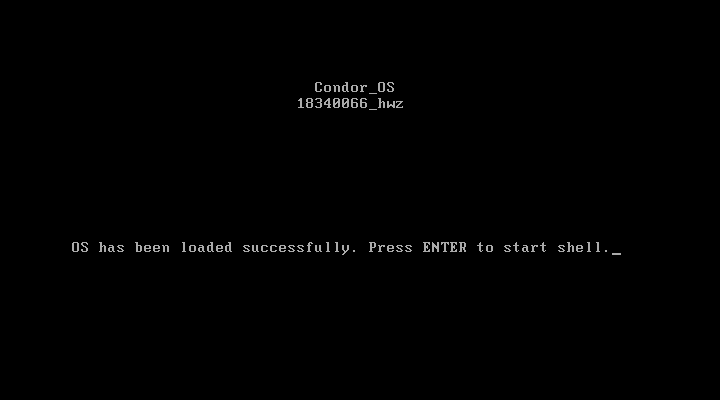
dd if=./temp/syscall\_test.bin of=Condor\_OS.img bs=512 seek=45 count=3 2> /dev/null

echo "Finished."

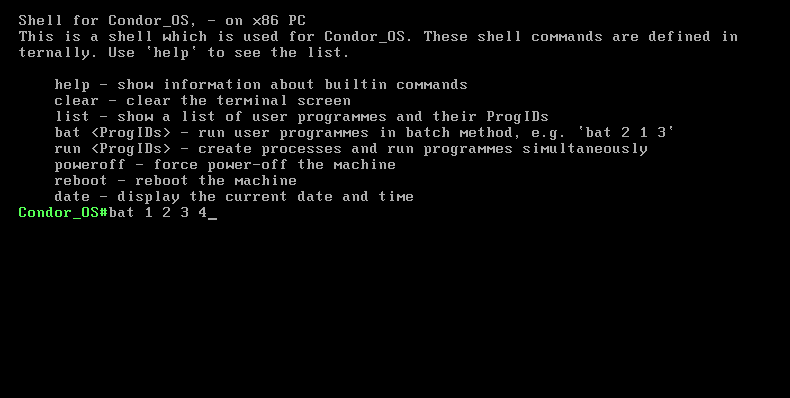


**实验结果：**

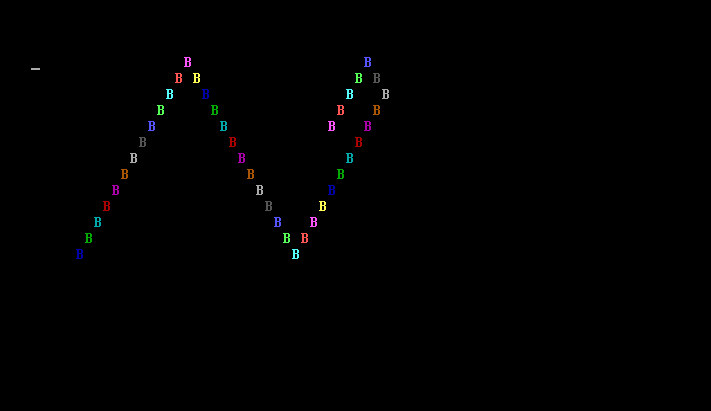
开机画面：

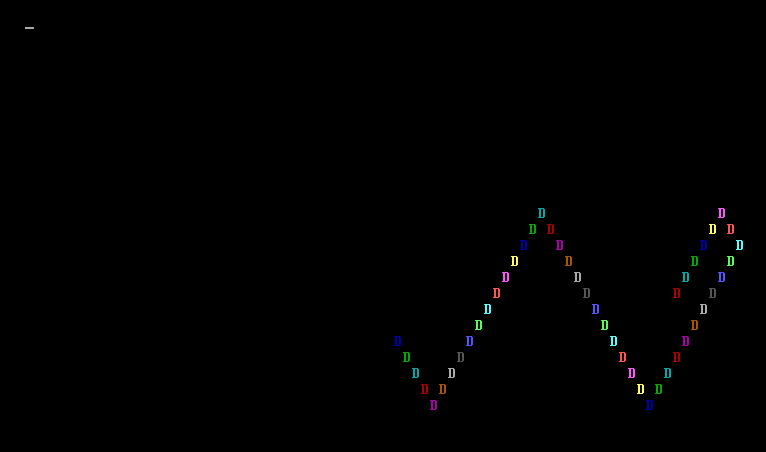


准备执行bat（批处理）命令：



展示结果（只展示首尾的两张效果图）：

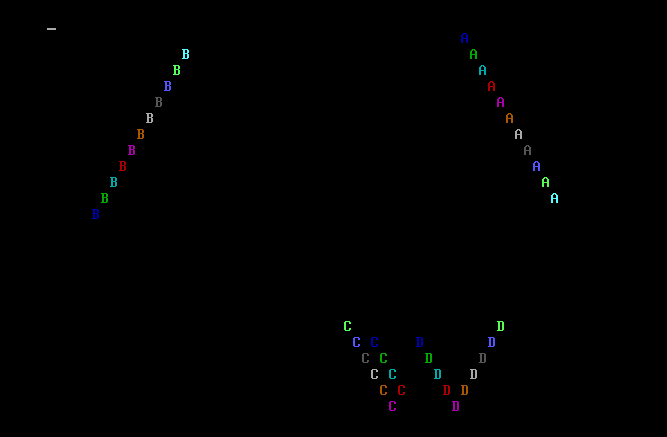




准备运行run命令（多线程并发执行用户程序）：



实验效果：



**实验体会：**

在本次实验中，我了解了多进程时间片轮转的工作原理：save→schedule→restart。而对这三个部分的实现，最为关键的就是对栈的各种操作。各种寄存器的保存，栈内元素的修改，也是需要注意的，否则就会出现难以察觉的错误。