课程设计报告

课　　程：操作系统原理课程设计

课设名称：在裸机保护模式下实现页机制和优先数调度机制的演示

院　　系：网络空间安全学院

专业班级：网安 2002 班

学　　号：U202012043

姓　　名：范启航

2023 年　03 月 3 日

目　录

1 课设目的

理解保护模式基本工作原理：保护模式是计算机操作系统用于实现多任务处理和内存保护的一种工作模式。它允许多个程序同时运行在计算机上，并提供了硬件级别的内存保护，防止一个程序意外地访问或破坏另一个程序的数据。通过理解保护模式基本工作原理，可以更好地设计操作系统管理进程、内存和设备等资源。

理解保护模式地址映射机制：在保护模式下，操作系统需要负责将每个程序使用的虚拟内存地址转换为对应的物理内存地址，并确保不同程序之间不会互相干扰。通过学习地址映射机制，可以更好地理解虚拟内存和物理内存的概念，以及操作系统如何管理它们。

段机制和页机制：段机制是一种内存保护机制，它将内存划分为多个段，每个段有自己的基地址和长度，并且可以设置访问权限。页机制是一种虚拟内存管理机制，它将虚拟内存划分为多个大小相等的页，并将它们映射到物理内存中的页面帧上。通过学习段机制和页机制，可以更好地理解保护模式下的内存管理和内存保护。

理解任务/进程的概念和切换过程：任务/进程是操作系统中的基本执行单元，每个任务/进程都有自己的代码、数据和执行状态。操作系统需要负责管理和调度多个任务/进程的执行，以及在它们之间进行快速的切换。通过学习任务/进程的概念和切换过程，可以更好地理解操作系统的调度算法和上下文切换机制。

理解优先数进程调度原理：优先数进程调度原理是一种常见的进程调度算法，它根据每个进程的优先级来决定下一个要运行的进程。优先级高的进程优先执行，而优先级低的进程则会被暂时放置在就绪队列中，等待调度器重新安排。调度器会将优先级最高的进程优先调度直至对应长度的时间片结束，后进行下一个进程调度，直至所有进程都已运行其对应长度的时间片。通过学习优先数进程调度原理，可以更好地理解进程调度的实现机制和调度算法的优化思路。

掌握保护模式的初始化：保护模式的初始化是操作系统实现保护模式的关键

步骤。在进入保护模式前，操作系统需要初始化各种系统数据结构（GDT 表，LDT 表）等、建立段表和页表、设置中断向量表、加载各类寄存器等等。掌握保护模式的初始化步骤和具体实现方法，以便能够理解操作系统的启动过程和内核的初始化过程。

掌握段机制的实现：段机制是保护模式下的一种内存管理机制，它通过将内存划分为多个段来实现内存保护，每个段具有不同的长度和特权级，每个段的选择子指向 GDT 或 LDT 内的描述符并以此来寻址。掌握段机制的实现原理和实现方法，以及如何使用段机制实现内存保护。

掌握页机制的实现：页机制是保护模式下的一种虚拟内存管理机制，它将虚拟内存划分为多个大小相等的页，并将它们映射到物理内存中的页面帧上。掌握页机制的实现原理和实现方法，以及如何使用页机制实现虚拟内存管理。

掌握任务的定义和任务切换：任务是操作系统中的基本执行单元，每个任务都有自己的代码、数据、堆栈、特权级和执行状态。掌握任务的定义和任务切换的实现原理，以及如何使用任务切换实现多任务处理。

掌握保护模式下中断程序设计：在保护模式下，中断是一种常见的操作系统交互方式。中断服务程序一般需要完成以下工作：保存中断前的现场、处理中断、恢复中断后的现场。在实现中断服务程序时，需要特别注意保护模式下的安全性问题，防止非法访问系统资源。可以在时钟中断服务中，对任务进行按照一定的算法机芯切换以此来实现任务调度的目的。掌握中断表中断概念和分类、熟悉中断向量表的概念和作用、实现基本的中断服务程序。

2 课程设计内容

启动保护模式，建立两个或更多具有不同优先级的任务（每个任务不停循环地在屏幕上输出字符串），所有任务在时钟驱动（时钟周期 50ms，可调）下进行切换。任务切换采用“优先数进程调度策略”。例如，设计四个任务，优先级分别为 16，10，8，6，在同一屏幕位置上各自输出：VERY，LOVE，HUST，MRSU 四个字符串，每个字符串持续显示的时间长短与他们的优先级正相关，体现每个任务的优先级的差异）。

3 程序设计思路

一个完整的操作系统通常由进入保护模式、GDT 的实现、任务 LDT 的实现、TSS 的实现、不同特权级任务切换、时钟中断和调度算法以及分页机制七个模块组成，下面我将依次介绍各个模块涉及到的概念、原理、设计和实现思路。

3.1 进入保护模式

在 IA32 下，CPU 有两种工作模式：实模式和保护模式。

- 实模式：在实模式下，CPU 只能访问 1MB 以下的物理内存，同时缺乏内存保护机制。
- 保护模式：在保护模式下，CPU 可以访问全部的物理内存，并且具备内存保护机制，保证不同进程间的数据互不干扰。

PC 启动时，CPU 是工作在实模式下，此时 CPU 有着 16 位的寄存器(Register)、16 位的数据总线(Data Bus)以及 20 位的地址总线(Address Bus)和 1MB 的寻址能力。一个地址是由段和偏移两部分组成，物理地址 = 段值 x 16 + 偏移。

进入保护模式后，拥有 32 位的寄存器(Register)和 32 位的地址总线，寻址空间达到 4GB。此时，地址仍然用段：偏移的形式来表示，不过段的概念发生了根本性的变化，其变成了一个索引指向一个描述符，而描述符中详细定义了段的起始地址、界限和属性等内容。

进入保护模式的设计和实现思路：

1) 实模式下，CPU 会从物理地址 0x0000:0x7c00（com 程序会在 0x0000:0x0100）处开始执行引导程序。
2) 引导程序首先会保存实模式下的栈指针的功能，保证在需要返回实模式时能够正确恢复现场。
3) 接下来会初始化 GDT（全局描述符表），对空描述符和代码段、数据段、堆栈段等描述符进行初始化。然后初始化各个任务的 LDT（局部描述符表），对代码段、数据段和堆栈段描述符进行初始化。
4) 加载 GDTR、IDTR 等寄存器的内容。
5) 打开 A20 地址总线，设置 CR0 寄存器的最高位（即 PE 位）为 1，进入保护

模式，跳转到保护模式对应的 32 位代码段中。

3.2 GDT 的实现

在操作系统中，GDT（全局描述符表）是一个数据结构，用于定义在保护模式下运行的程序所使用的内存段，提供段式存储机制，包括代码段、数据段、堆栈段等。GDT 是操作系统中实现内存保护和虚拟内存的关键组成部分。

段描述符：GDT 中的每个表项都是一个段描述符，用于描述一个内存段的基址、大小、访问权限和其他属性。组成如下图 3-1 所示：

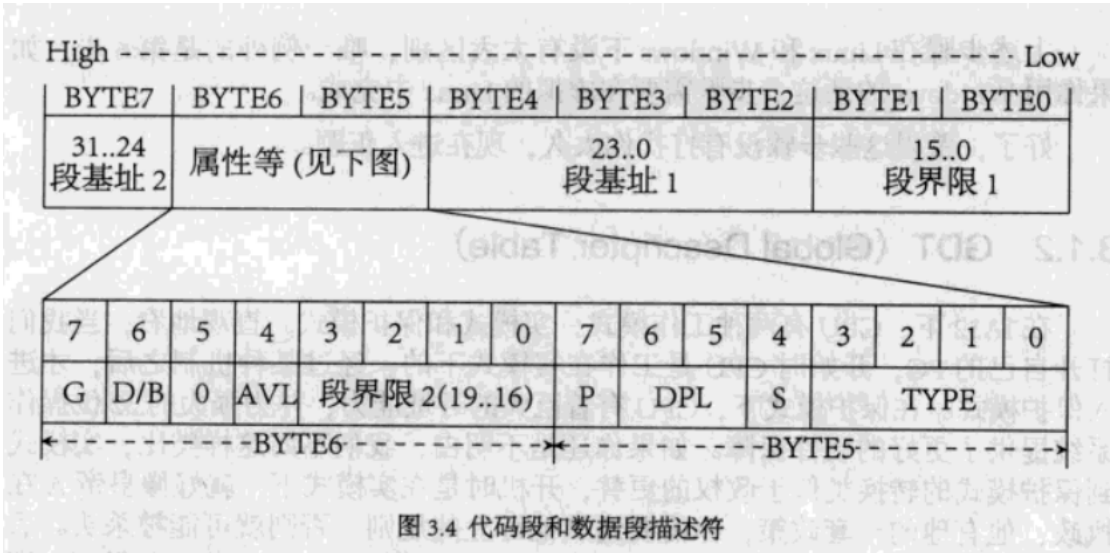

图 3-1 段描述符结构

选择子：保护模式下，选择子是用来访问内存段的句柄，由两个部分组成：索引和特权级。组成如下图所示：

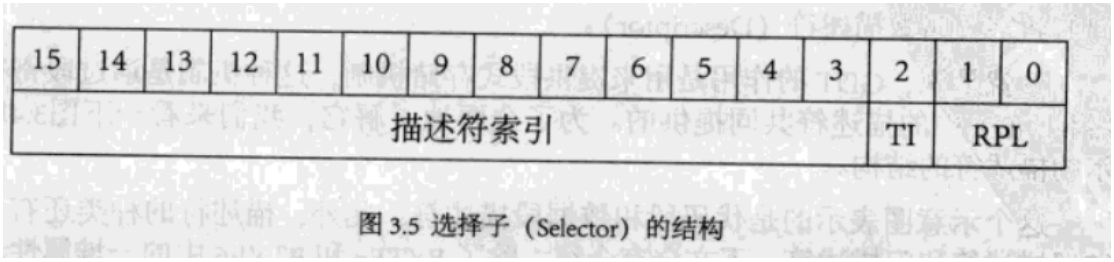

图 3-2 选择子结构

GDT 表寄存器（GDTR）：用于存储 GDT 的基地址和表长，CPU 在访问 GDT 时需要使用该寄存器。组成如下图所示：

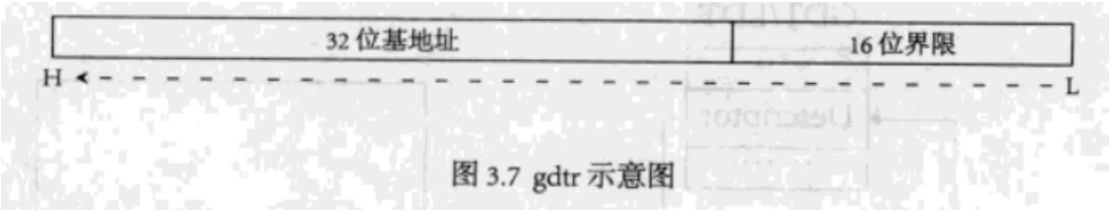

图 3.7 gdtr 示意图

图 3-3 gdtr 结构

GDT 表的设计与实现思路：

1) 定义 GDT 段，在 GDT 段内定义 32 位代码段、16 位代码段、VIDEO 段、数据段、堆栈段、任务的 LDT 段以及 TSS 段描述符。

2) 定义 GDT 段描述符的选择子。

3) 在进入保护模式前对 GDT 段描述符进行初始化，初始化其基址，长度等属性。

4) 加载 GDT 段寄存器： lgdt [GdtPtr]

3.3 任务 LDT 的实现

LDT 与 GDT 类似，都是段描述符表，区别仅仅在全局和局部的不同。在本实验任务中，一个任务的 LDT 包括一个 ring3 的任务段，ring3 的数据段，ring3 的堆栈段和一个 ring0 的堆栈段。

与 GDT 不同的是，LDT 的描述符位于 GDT 内，而 LDT 的其他段描述符位于一个单独描述表内。该局部描述符表通过 LDT 的选择子寻址。具体寻址过程：通过 LDT 选择子寻址 LDT 描述符表，通过段选择子，寻址段的基址。

段描述符、选择子概念与 GDT 相同，但 ldtr 寄存器与 gdtr 寄存器不同，为 16 位选择子。

LDT 的设计与实现思路：

1) 在 GDT 表中定义 LDT 段描述符与选择子。

2) 定义 LDT 段，定义任务所需的段描述符与局部段选择子。

3) 在切换任务时，加载 LDT 到 LDTR 寄存器内，以便处理器能够正确寻址。

3.4 TSS 的实现

TSS 是一种数据结构，用于保存处理器运行过程中所需的任务状态信息，包括任务的堆栈指针、程序计数器和处理器寄存器等。TSS 是在操作系统内核初始化时被定义的，并在进程创建时由操作系统为每个进程分配。

TSS 描述符：位于 GDT 内，描述了 TSS 的位置和大小。

TSS 段结构如下图所示：

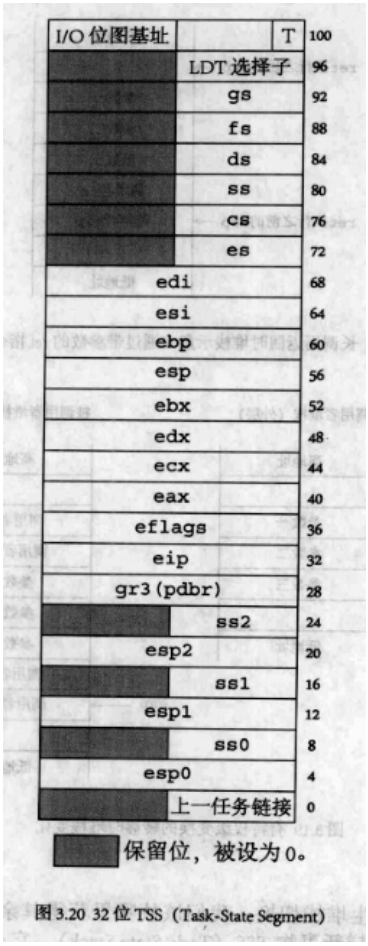

图 3-4 TSS 段结构图

在进程创建时，操作系统会为该进程创建一个独立的 TSS，该 TSS 会包含该进程的所有任务状态信息。每个进程的 TSS 都是相互独立的，这样就可以保证不同进程之间的任务状态相互隔离。当进程被调度执行时，处理器会从该进程的 TSS 中获取任务状态信息，以便正确地执行该进程的程序。

TSS 的设计与实现思路：

1) 初始化 TSS 段：将 TSS 段的任务状态信息初始化为默认值。
2) 修改 TSS：进程可以通过修改自己的 TSS 来控制自己的任务状态信息。这可以通过修改 TSS 中的任务状态信息实现。
3) 切换 TSS：当进程切换时，需要将进程的 TSS 加载到处理器中，以便处理器能够正确地获取该进程的任务状态信息。

3.5 不同特权级任务切换

由于时钟中断处于 ring0 特权级态，任务运行于 ring3 特权级态，在进行任务调度时需要进行特权级切换操作， 使用 TSS 和 iretd 指令进行切换。

特权级：操作系统中通常会将处理器的权限划分为多个级别，称为特权级。这些特权级通常包括内核态和用户态，不同的特权级对应不同的系统资源访问权限。

不同特权级任务切换的设计与实现思路：

1) 加载任务对应的 TSS。
2) 加载任务的 LDT 段描述符
3) 修改 ds 寄存器为任务的数据段
4) 使用 iretd 进行任务切换

3.6 时钟中断和调度算法

时钟中断和调度算法是实现多任务处理的重要机制，涉及到的概念包括时钟中断、中断处理程序、调度算法。

时钟中断：由操作系统设置的硬件定时器产生的，用于定期发出中断请求。通常，操作系统会将时钟中断的间隔时间设置为几毫秒或几十毫秒。在本实验中，使用 8259A 可编程中断控制器，能够产生可屏蔽中断并传递给 CPU 处理，8259A 的结构如下图所示：

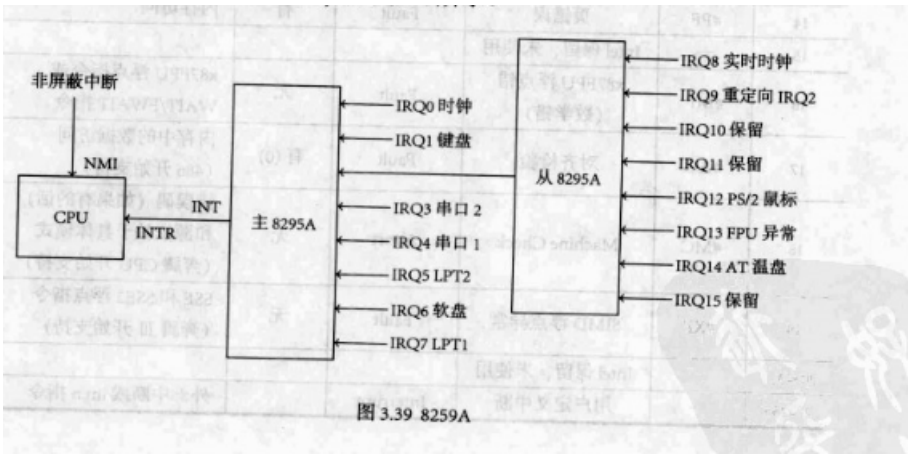

图 3-5 8259A

中断处理程序：当操作系统收到时钟中断请求时，会触发中断处理程序。中断处理程序会将当前进程的状态保存到内存中，然后将控制权交给调度器。

调度算法：调度算法是决定哪个进程应该被执行的一组规则。在本实验中，我们选择优先数进程调度策略，优先选择优先级最高的任务进行调度，直至其时

间片用完后，再调度下一个任务。

实现时钟中断和调度算法的设计与实现思路：

1) 设置硬件定时器 8259A，通过对其设置，使其定时产生时钟中断请求。

2) 建立 IDT 表，设置各个中断向量对应的描述符与选择子，加载 IDTR。

编写时钟处理程序，在时钟中断处理程序中写入调度算法与任务调度，实现根据优先数进行任务调度的效果。

3) 在初始化完成后打开时钟中断响应，等待时钟中断到来。

3.7 分页机制

分页机制是一种虚拟内存管理技术，它涉及到的概念包括页表、页目录、页表项、页大小和页面置换算法等。在未打开分页机制时，线性地址等于物理地址。但当开启分页时，情况发生变化，分段地址将逻辑地址转换成线性地址，线性地址再通过分页机制转换成物理地址，如下图所示：

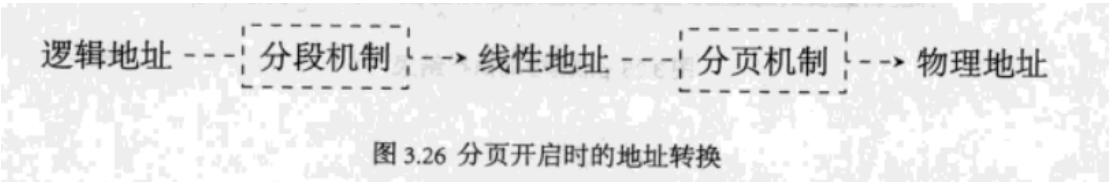

图 3.26 分页开启时的地址转换

图 3-6 分页机制

页表：页表是一种数据结构，用于将虚拟地址映射到物理地址。在分页机制中，每个进程都有自己的页表，它包含了虚拟地址和对应的物理地址之间的映射关系。

页目录：页目录是一种数据结构，用于存储页表的地址。在分页机制中，每个进程都有一个页目录，它指向该进程的页表。

页表项：页表项是页表中的一个条目，包含了虚拟地址和对应的物理地址之间的映射关系，以及一些控制位，如访问权限、脏位和有效位等。

页大小：页大小是指操作系统中一次分配的内存块的大小。常见的页大小包括 4KB、8KB 和 16KB 等。

cr3：高 20 位是页目录表首地址的高 20 位，页目录首地址的低 12 位会是 0，页就是说页目录表是 4KB 对其的。结构如下图所示：

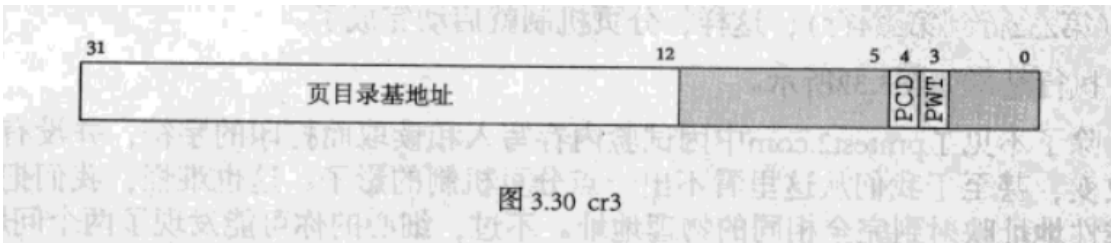

图 3.30 cr3

图 3-7 cr3 结构

实现分页机制的设计与实现思路：

1) 设置每一个任务的页目录和页表的起始地址。
2) 初始化每一个任务的页目录与页表，为方便期间，我们假定线性地址=物理地址，一一对应。
3) 在进入保护模式后，启动分页机制。
4) 在切换任务的同时，切换 cr3 为其对应的页目录基地址。

4 实验程序的难点或核心技术分析

4.1 实验流程总结

本实验完成了内容 1,2,3,4 的所有内容，分别对应 task1,task2,task3,task4。下面主要总结实验内容 4 的流程。

在内容 4 中，需要定义多个任务，每个任务的优先数不同，在时钟中断服务程序中，采用优先数调度算法，实现多个任务的切换。其中每个任务时死循环，循环输出字符串。每个任务字符显示时长与优先级一致（优先数大，运行机会多，显示时长就大）时钟中断服务程序（选择任务，切换任务）。实验过程大致内容如下：

1. 创建 GDT 表，对 GDT 表内的段描述符和选择子进行初始化，包括段描述符的基址，界限以及各种属性、段选择子的偏移和属性。
2. 创建 IDT 表，将中断向量与其对应的门选择子与偏移对应。
3. 初始化 TSS 段，对 TSS 段进行初始化。
4. 创建任务 LDT 表，对 LDT 表内任务数据段、代码段和堆栈段等描述符和选择子进行初始化。
5. 对段描述符进行初始化，在 16 位代码段，将段基址装入段描述符内。
6. 对不同任务页目录表和页表初始化。
7. 初始化 8259A、8253A 和 IDT，在执行过程中设置 IRQ0 配置时钟中断，同时配置时钟中断的周期。
8. 编写时钟中断调度程序和调度算法，选择此时应调度的任务。
9. 通过不同特权级的跳转切换到对应的任务，循环输出任务的字符串。

4.2 实验重难点

本书完成了课设任务书要求的任务 4 的全部内容，其中的主要难点在任务的定义，跨特权级的任务调用与调度算法的实现。

1. 任务的定义：

由于每个任务的 LDT 段、代码段、数据段和堆栈段的定义大致相同，多为重复性动作。因此我使用了汇编的宏定义 DefineTask 4，并添加了 4 个参数来区

分各个不同的任务。其中参数 1 为任务号，2 为输出字符串内容，3 为输出代码的行数，4 位输出代码的颜色。任务定义宏如下所示：

```
; 定义任务
; usage: DefineTask num, string, row,color
%macro DefineTask 4
    DefineLDT     %1
    DefineTaskCode   %1, %3, %4
    DefineTaskData   %1, %2
    DefineTaskStack0      %1
    DefineTaskStack3      %1
    DefineTaskTSS     %1
%endmacro
```

而对于每个任务的代码 LDT，代码段，数据段，堆栈段和 TSS 均使用宏进行构建，大大减少了冗余的代码量，且结构也更为清晰，调试查错也更加方便。举例代码段如下所示：

```
; 定义任务代码段
; usage: DefineTaskCode num, row, color
%macro DefineTaskCode 3
[SECTION .task%1code]
ALIGN     32
[BITS     32]
LABEL_TASK%1_CODE:
; 遍历输出 szTask0Message
    xor       ecx, ecx
    mov       ah, %3
.outputLoop:
    mov       al, [szTask%1Message + ecx]
    mov       [gs:((80 * %2 + ecx) * 2)], ax
    inc       ecx
```

```
    ; 判断是否到达字符串末尾
    cmp     al, 0
    jne     .outputLoop
    jmp     LABEL_TASK%1_CODE
Task%1CodeLen   equ $ - LABEL_TASK%1_CODE
; END of [SECTION .task%1code]
%endmacro
```

2. 跨特权级的任务调用：

任务调用部分设计到特权级的变换，因此不能简单的使用 jmp 或 call 进行跳转。在 x86 架构下，使用 IRETD 指令可以进行特权级变换，即从低特权级到高特权级的变换。IRETD 指令主要用于从中断或异常处理程序返回到用户态代码，并且将控制权交给一个新的代码段和堆栈。

(1) 中断或异常处理程序通过执行 IRETD 指令返回到用户态代码。IRETD 指令会弹出堆栈中的 EIP 寄存器值，该值指向新的代码段中的指令。

(2) IRETD 指令同时还会弹出 CS 寄存器的值，该值是指向新的代码段选择器。这个选择器包含新的代码段的描述符在 GDT 表中的偏移量。

(3) RETD 指令还会从堆栈中弹出 EFLAGS 寄存器的值，并将其写入 EFLAGS 寄存器中。这是因为中断或异常处理程序可能已经更改了标志寄存器的值。

(4) 最后，IRETD 指令还会从堆栈中弹出一个新的堆栈选择器和堆栈指针，这是为了切换到新的堆栈。

因此在使用 IRETD 进行任务调用跳转时，我们需要依次将任务对应的 SS、ESP、EFLAGS（开启中断后），CS 和 EIP 压入堆栈中，以此来正确寻址。下面的代码中同样使用了宏，代码如下：

```
    push    SelectorTask%1Stack3    ; SS
    push    TopOfTask%1Stack3   ; ESP
    pushfd                      ; EFLAGS
    pop     eax                 ; ┐
    or      eax, 0x200          ; ├ 将 EFLAGS 中的 IF 位置 1, 即开启中
```

断

```
push    eax                 ;┘
push    SelectorTask%1Code  ; CS
push    0                   ; EIP
```

3. 调度算法

在本任务中，任务调度算法属于重难点，根据任务书的要求，具体的任务优先数调度算法如下：在进程创建时有一个初始的优先数，每次调度时，会先判断前一个任务的 ticks 是否为 0，若不为 0 继续执行该任务，若为 0，则在其他任务中选择 ticks 最高的任务。若所有的任务 ticks 均为 0，则进入新的一轮调度，将每个任务的优先数赋给 ticks。如下流程图 4-1 所示：

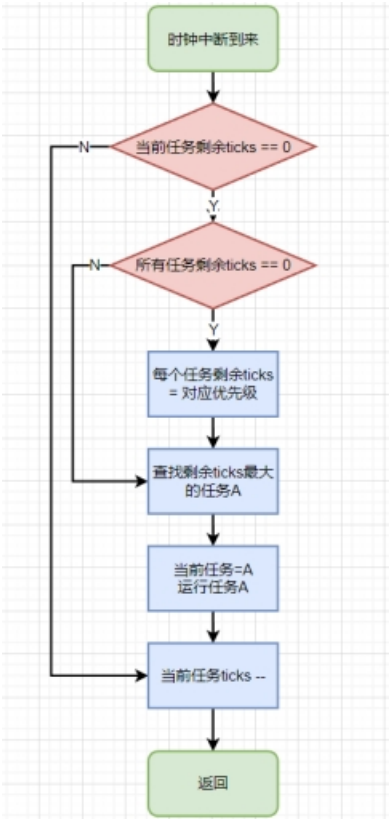

图 4-1 调度算法流程图

上述的算法看上去比较简单，但使用汇编语言实现较为困难，还需要考虑寄存器，程序控制等细节，因此我选择先使用 C 语言编写，再使用工具将其转换为汇编语言。C 语言实现如下所示：

```c
int RunningTask = 0;
int LeftTicks[] = {0, 0, 0, 0};
int TaskPriority[] = {16, 10, 8, 6};
int main()
{
    if(LeftTicks[RunningTask] == 0)
    {
        if(LeftTicks[0] == 0 && LeftTicks[1] == 0 &&
        LeftTicks[2] == 0 && LeftTicks[3] == 0)
        {
            for(int i = 0; i<4; i++)
            {
                LeftTicks[i] = TaskPriority[i];
            }
        }
        int max = 0;
        for(int i = 0; i<4; i++)
        {
            if(LeftTicks[i] > max)
            {
                max = TaskPriority[i];
                RunningTask = i;
            }
        }
    }
    LeftTicks[RunningTask]--;
}
```

图 4-2 调度算法 C 语言

将其使用 gcc 转换为 main.o，再使用 objconv 工具将 main.o 转换成 nasm 汇编代码，将其数据部分进行稍微修改后，即可直接插入时钟中断响应函数中。汇编代码较长，不在此展示。

5 开发和运行环境的配置

开发环境：Windows10，vscode 编辑器，NASM，DiskGenius

运行环境：Windows10，Bochs x86 Emulator 2.7，freedos

配置文件：

megs: 32

romimage: file="C:\\Program Files\\Bochs-2.7\\BIOS-bochs-latest"

vgaromimage: file="C:\\Program Files\\Bochs-2.7\\VGABIOS-lgpl-latest"

floppya: 1_44=freedos.img, status=inserted

floppyb: 1_44=pmtest.img, status=inserted

boot: a

mouse: enabled=0

magic_break: enabled=1

panic: action=report

display_library: win32, options="gui_debug"

更新程序：在 windows 上，通过 NASM 编译为.com 文件，使用 DiskGenius 打开 pmtest.img 软盘，将程序移入即可

运行程序：在 windows 上，通过 bochs 打开 freedos，直接运行程序。

调试程序：在程序中添加断点 xchg bx, bx。将配置文件改为 bocchsrcWin.bxrc 并修改配置文件，右键即可直接 debug，如下图 5-1 所示

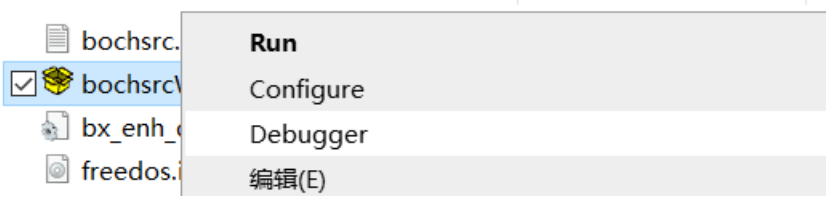

图 5-1 调试代码

通过观察屏幕上的输出内容来判断程序的任务切换状态。

6 运行和测试过程

Ready 表示程序已初始化完成，VERY 为第一个任务的输出，由于其优先数最大，运行时间最长。如下图 6-1 所示

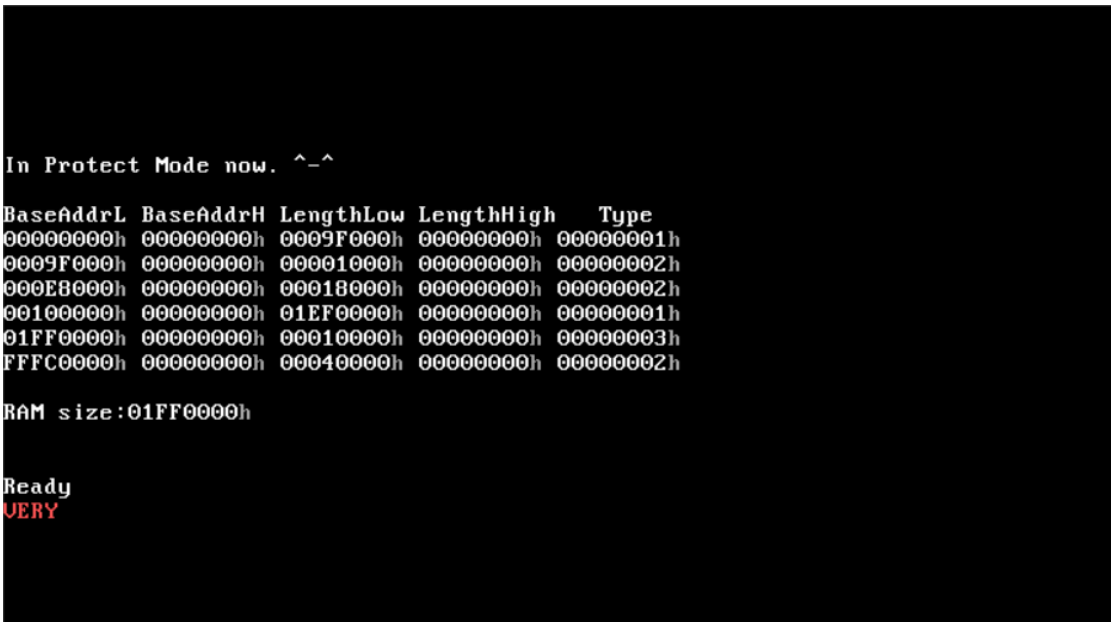

图 6-1 第一个任务

LOVE 为第二个任务的输出，由于其优先数第二大，运行时间第二长。

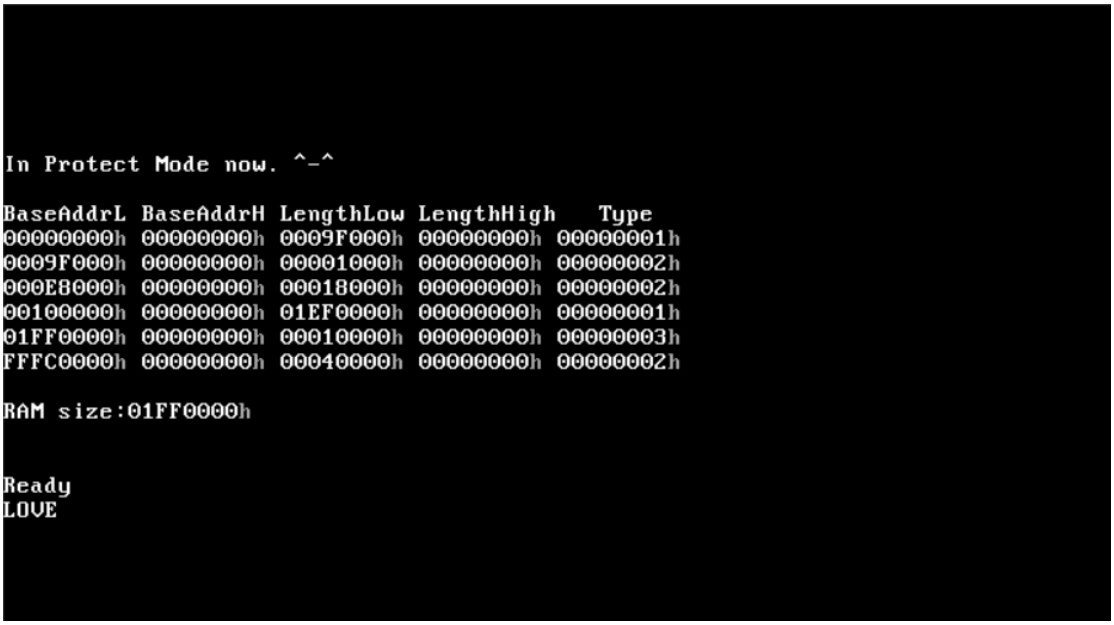

图 6-2 第二个任务

HUST 为第三个任务的输出，由于其优先数第三大，运行时间第三长。如下

图 6-3 所示

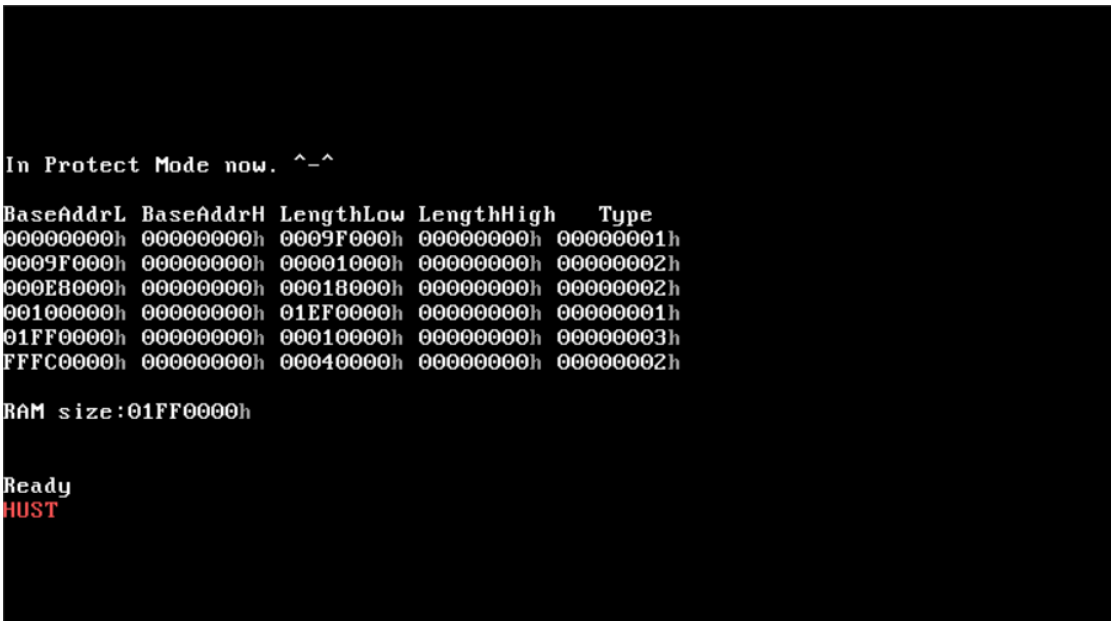

图 6-3 第三个任务显示图

MRSU 为第四个任务的输出，由于其优先数最小，运行时间最短。如下图 6-4 所示。

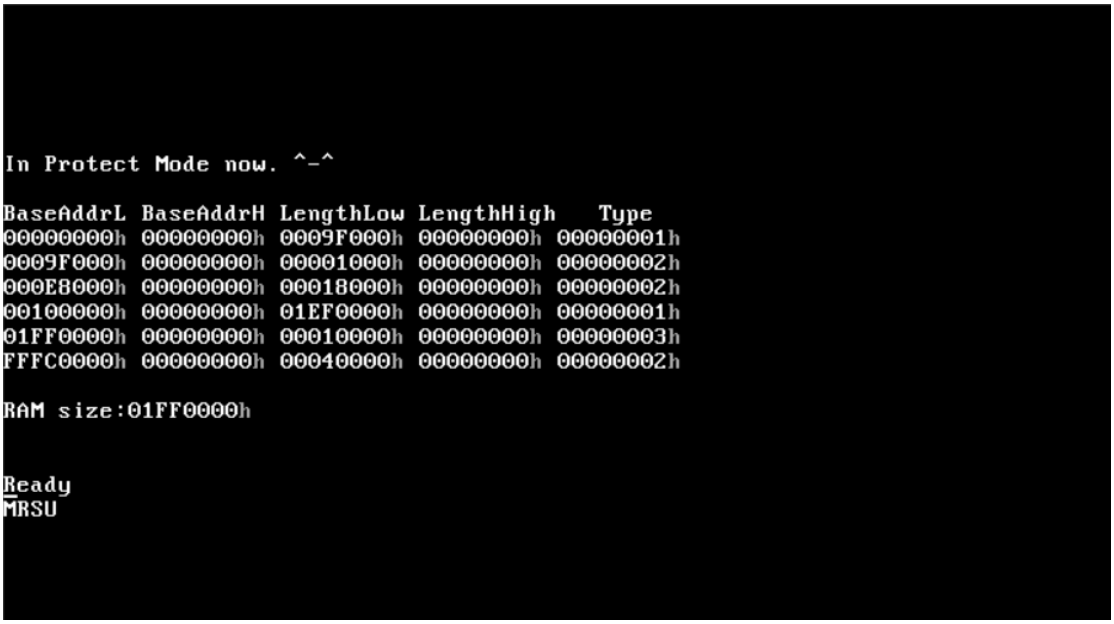

图 6-4 第四个任务显示图

7 实验心得和建议

在本次实验中，我成功实现了一个带有优先数调度算法的多任务操作系统，通过这个实验，我收获了许多关于操作系统原理的知识，并且也遇到了一些问题。问题及解决方法如下：

(1) 问题：Linux 下 bochs 虚拟机开机后黑屏

解决方法：更换使用了 Windows 作为开发环境，后来得知是 Ubuntu 版本号问题，Ubuntu22.04 无法使用 APT 版本，需要从源码编译后使用。

(2) 问题：Windows 开发环境下无 mount 的指令挂载软盘

解决方法：使用 DiskGenius 工具即可打开软盘进行挂载。

(3) 问题：参考代码 pmtest6,7,8 闪退，无法看到运行效果

解决方法：给的参考代码在退出保护模式时，只关闭了保护模式，没有关闭分页机制，添加如下代码即可解决：

```
mov eax, cr0
and eax, 7fffffffh
mov cr0, eax
```

(4) 问题：Ring3 代码段无法在屏幕上输出内容

解决方法：将 VIDEO 段加上 DPL3 属性，使 Ring3 代码段也能调用

(5) 问题：LABEL，选择子经常会打错或漏写

解决方法：根据 NASM 报错内容进行改正。

(6) 问题：初始化段描述符时，代码重复存在冗余，不利于人编辑和调试。

解决方法：使用宏进行编写，宏如下：

```
; 初始化段描述符
; usage: InitDescBase LABEL, LABEL_DESC
%macro InitDescBase 2
    xor eax, eax
    mov ax, cs
    shl eax, 4
```

```
    add eax, %1
    mov word [%2 + 2], ax
    shr eax, 16
    mov byte [%2 + 4], al
    mov byte [%2 + 7], ah
%endmacro
```

(7) 问题：任务 2 中，从 ring3 的代码段跳转到另一个 ring3 的代码段时存在错误

解决方法：添加一个调用门，先从 ring3 的代码段跳转到 ring0 的调用门中，再进入另一个 ring3 的代码段。

(8) 问题：调度算法使用汇编较难实现，一直存在 bug

解决方法：使用 C 语言编写，再将其转为汇编语言

(9) 问题：若程序在 32 位代码段中陷入死循环，则在时钟调度时会产生 GDT 越界错误

解决方法：暂时未找到原因，不过如果在 32 位代码段中进入某个任务后，则调度时不会产生此问题，待解决。

8 学习和编程实现参考网址

（1）《操作系统原理》教材…

（2）《自己动手写一个操作系统》…

（3）课件