课程设计报告

课　　程：操作系统原理课程设计

课设名称：在裸机保护模式下编写多任务并演示页机制和优先数调度机制

院　　系：网络空间安全学院

专业班级：网安 2104 班

学　　号：U202112131

姓　　名：邬雪菲

2024 年　04 月 07 日

目　录

1 课设目的 1

2 课程设计内容 2

3 程序设计思路 3

4 实验程序的难点或核心技术分析 11

5 开发和运行环境的配置 17

6 运行和测试过程 18

7 实验心得和建议 20

8 学习和编程实现参考网址 22

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的一种工作模式，旨在实现多任务处理和内存保护。其核心原理在于允许多个程序同时在计算机上运行，并通过硬件级别的内存保护机制，防止数据被意外访问或破坏。它使操作系统能够有效管理进程、内存和设备等资源，并确保每个运行的程序都处于隔离环境中，不受其他程序的干扰。这种隔离性不仅保证了系统的稳定性和安全性，还为多任务处理提供了坚实的基础，使得用户能够同时运行多个程序，而不必担心彼此之间的冲突或干扰。因此，理解保护模式的基本工作原理对于设计和实现现代操作系统至关重要。

理解保护模式地址映射机制：保护模式的地址映射方式和实模式有所不同，这种不同在提升计算机寻址能力的同时也使得内存资源的管理更有效和安全。通过学习保护模式地址映射机制，能够更好地理解内存映射和管理机制，加深对操作系统的了解。

段机制和页机制：是操作系统中用于内存管理的两种关键技术。段机制将内存划分为若干段，每段具有独立的特性和权限；页机制将内存划分为大小相等的页，程序按页加载。这两者提供了对内存的有效管理和保护，确保程序安全运行、资源高效利用，同时提供了灵活性和可扩展性，促进了多任务处理。

理解任务/进程的概念和切换过程：任务是程序的执行实例，包括代码、数据和执行状态。进程是任务的抽象，代表了操作系统进行资源分配和调度的基本单位。进程切换是指在多任务系统中，操作系统暂停当前运行的进程并切换到另一个进程的过程。切换涉及保存当前进程的状态、恢复下一个进程的状态，并更新操作系统的调度信息。这个过程保证了系统资源的合理利用和任务的及时响应，是操作系统实现多任务处理的关键机制之一。

理解优先数进程调度原理：不同进程调度方式影响着计算机执行多任务的流程。优先数进程调度赋予每个进程优先数来表示优先程度，优先级高的进程会先运行。这种调度原理保证了对关键任务的及时响应，并优化了系统资源的利用效率，提高了系统的性能和响应能力。

掌握保护模式的初始化：操作系统需要完成各种数据结构（如 GDT、LDT、IDT 等）、控制寄存器等等的设置，然后才能进入保护模式。掌握保护模式的初始化是操作系统开发和调试的基础，为构建稳健、高效的系统奠定了坚实的基础。

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，它将内存分割成多个段来实现内存保护，每个段具有不同的长度和权限级别并通过对应的段描述符来保存段的大小、基址和各属性信息。通过段机制，操作系统可以控制对内存的访问，并实现内存的安全性和隔离性。

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页并映射到物理内存中的页面帧上，使得连续的虚拟内存可映射到不同的物理内存处。掌握页机制的实现原理和实现方法，并使用页机制实现虚拟内存管理是关键点。

掌握任务的定义和任务切换：任务是程序的执行实例，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是课设的关键点。

掌握保护模式下中断程序设计：在保护模式下，中断处理程序需要通过中断描述符表（IDT）来识别和处理中断。操作系统需要初始化 IDT，并将中断处理程序的地址注册到相应的中断向量中。当硬件触发中断时，处理器会根据中断向量查找 IDT，并跳转到对应的中断处理程序。在中断处理程序中，操作系统需要保存当前进程的状态，执行中断服务例程，并根据需要进行进程调度。最后还需恢复进程状态并返回中断点继续执行。需要掌握熟悉中断向量表的概念和作用、实现基本的中断服务程序，这是本课设的重点之一。

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

操作系统的多任务调度依赖保护模式下的段页式寻址方式、中断处理和各项数据结构，由此我将程序设计分为以下六个模块：实验环境配置、数据结构定义、实模式下的准备、保护模式初始化、分页机制启动、时钟中断与任务调度。下面是整个程序设计的流程图。

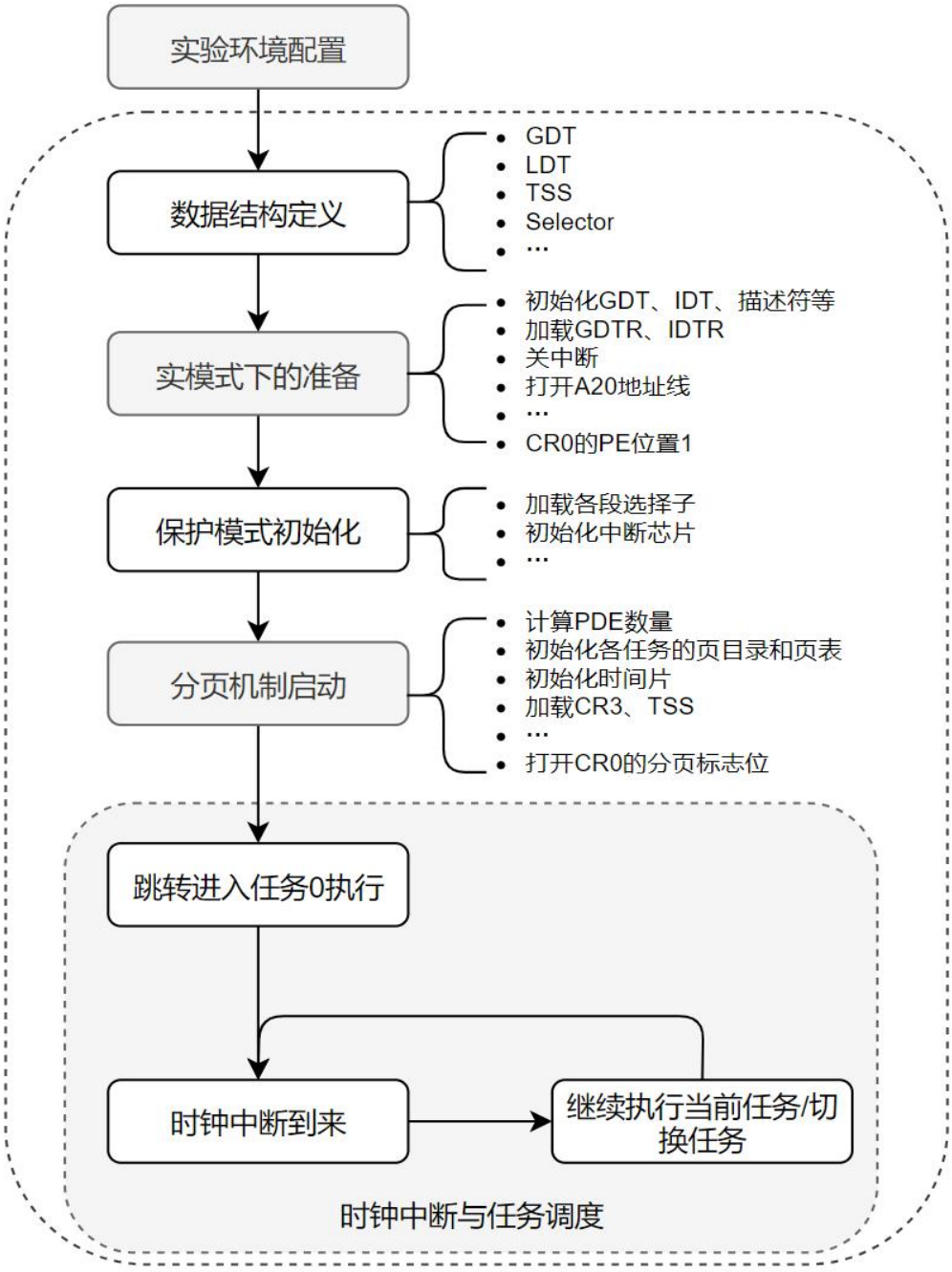


图 3-1 程序设计流程图

3.1 实验环境配置

本次实验在 VMware Workstation 中安装 ubuntu，并使用 Bochs 虚拟机来模拟程序的执行。具体步骤如下：

1）安装 nasm 和 bochs 虚拟机。

2）准备 freedos 软盘映像文件。由于引导扇区有着 512 字节的限制，当程序过大时无法直接写入执行，需要编写一个类似操作系统内核的引导扇区来读取程序并运行。但此步骤的难度较大，于是本实验改为借助现有的 DOS 来执行程序。

3）制作用于存储测试程序的空白盘映像文件，并挂接到目录中便于频繁更新编写的实验程序。

4）编写 bochs 配置文件并启动 bochs 测试。

3.2 数据结构定义

操作系统的保护模式有着更大的寻址能力，并为 32 位操作系统提供了更好的硬件保证，这是在依赖 GDT、LDT、TSS 等数据结构的情况下实现的更有效的内存管理。要进入保护模式，必须对各数据结构进行合理的初始化。本次实验要求实现多任务调度，除了设置全局描述符表，还需要为每个进程都设置一个局部描述符表存储每个进程的任务段信息，同时由于存在特权级切换，还需要设置 TSS 任务状态段结构。

1）GDT：全局描述符的作用是提供段式存储机制，这种机制是通过段寄存器和 GDT 中的描述符共同提供的。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，结构如下图所示：

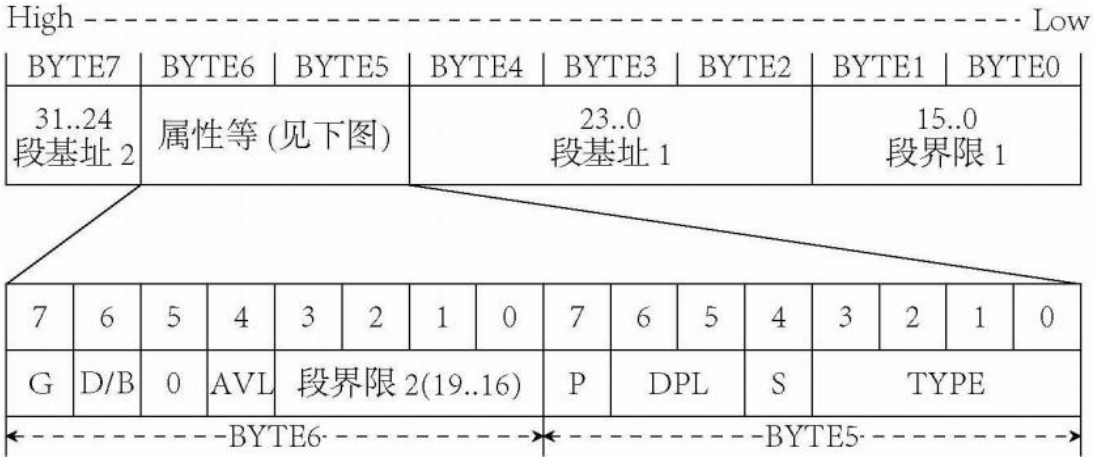


图 3.2-1 段描述符结构

2）Selector：选择子是保护模式下用来访问内存段的句柄，由两个部分组成：索引和特权级，其结构如下图所示：

图 3.2-2 选择子结构

3）LDT：局部描述符与 GDT 类似，都是段描述符表，区别仅仅在于局部和全局。本实验中每个任务的 LDT 包括：ring3 数据段，ring3 代码段，ring0 堆栈段，ring3 堆栈段，任务状态段 TSS。且与 GDT 不同的是，LDT 自身的描述符放在 GDT 内，而任务的其他段描述符放在 LDT 内。寻址时由 LDT 选择子确定 GDT 中的 LDT 描述符，再通过段选择子在 LDT 中确定段描述符后定位段基址。寻址过程如下图所示：

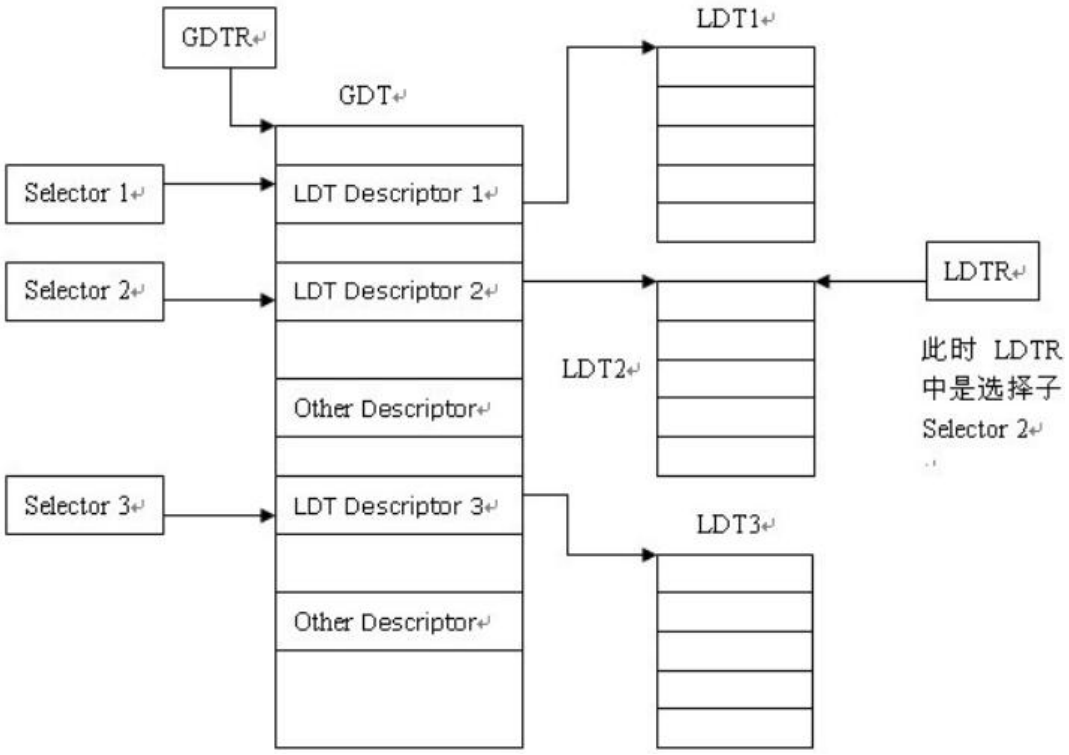


图 3.2-3 LDT 寻址

4）Gate：门也是一种描述符，一个门描述了由一个选择子和一个偏移所指定的线性地址，即定义了目标代码对应段的选择子、入口地址的偏移和属性等。但其结构和普通的段描述符不同，不过第 5 个字节都表示属性，如图 3.2-4 所示：

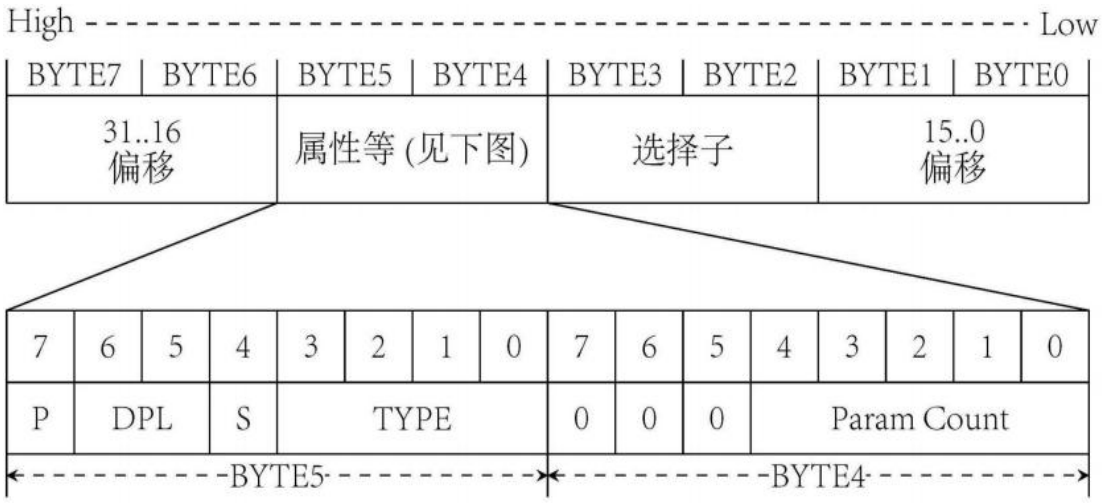


图 3.2-4 门结构

中断门是一种特殊的调用门，调用门与 call 或 jmp 指令结合使用可以实现不同特权级代码的转移。中断门用于中断处理程序的定位与调用，其中 20h 中断专用于时钟中断处理。中断处理寻址过程如图 3.2-5 所示：

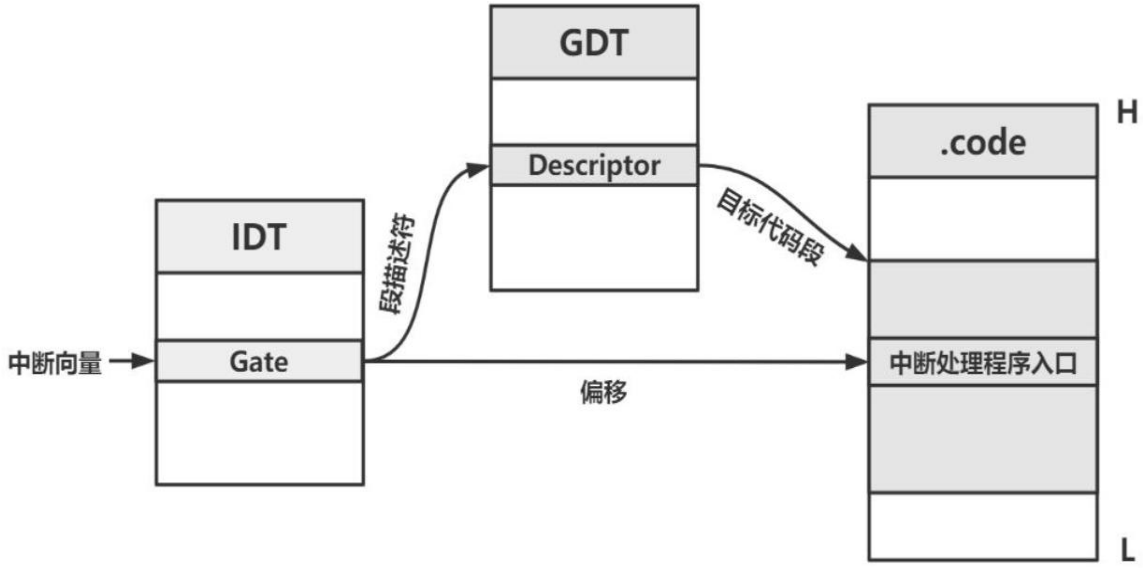


图 3.2-5 中断处理寻址

5）TSS：任务状态段是保护模式下用于保存处理器运行过程中任务状态信息的一种数据结构，它包含了任务的特权级别、寄存器内容、描述符指针等信息。当切换特权级执行任务时，会使用 TSS 来保存当前的任务状态，并加载下一个任务的 TSS，以实现任务的无缝切换。其结构如图 3.2-6 所示：

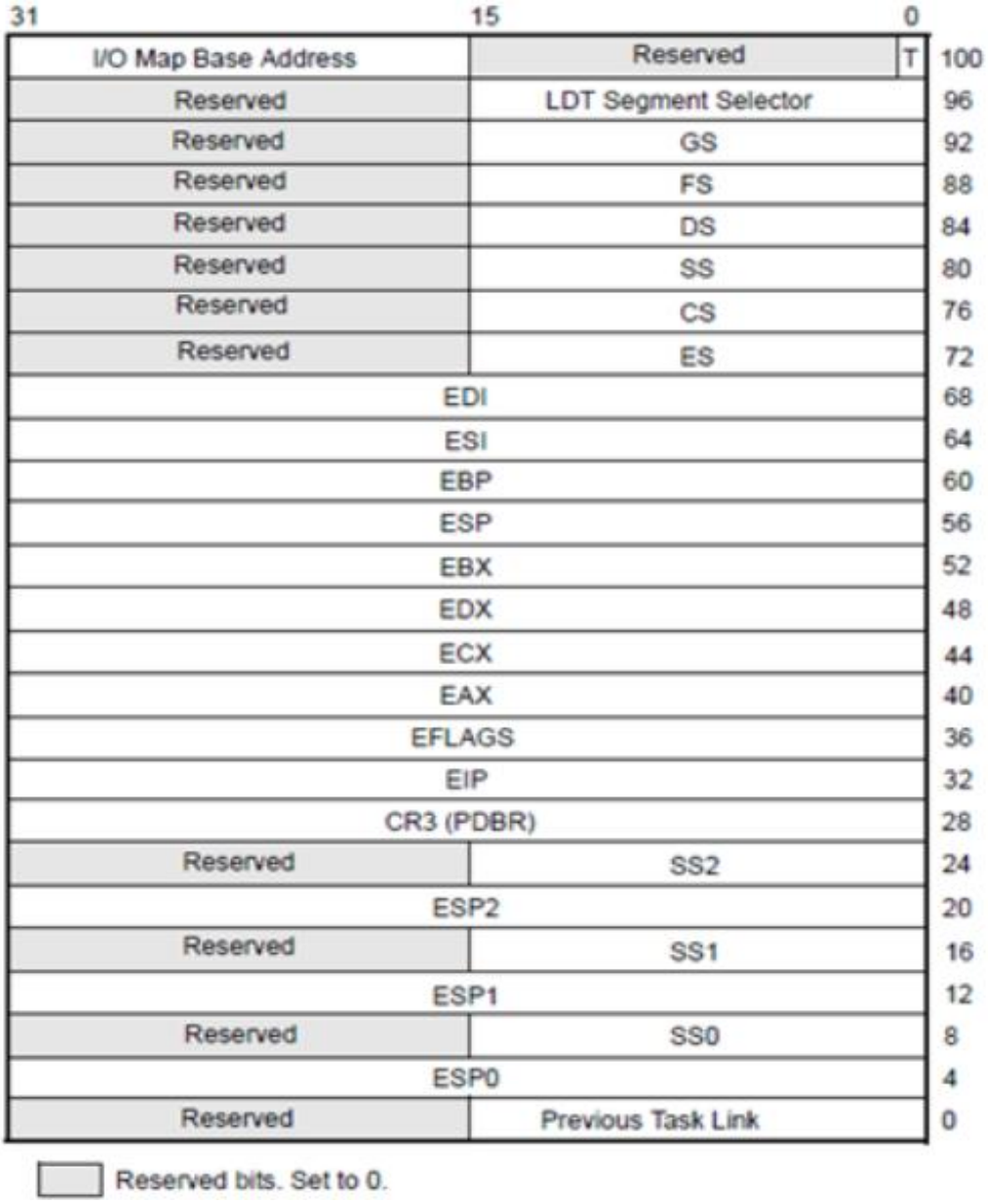


图 3.2-6 TSS 结构

在本实验中需要完成四个任务的调度切换，所以需要定义四个 LDT，四个 TSS 和四套页表，并通过时钟中断来切换任务，故需要修改中断向量表中 20h 号的时钟中断处理门。

3.3 实模式下的准备工作

CPU 有两种工作模式：实模式与保护模式，计算机启动后处于实模式下，有着 16 位的寄存器、16 位的数据总线以及 20 位的地址总线，可以访问 1MB 的物理内存，地址=段值*16+段内偏移。但是实模式下缺乏内存保护机制，必须要切换到保护模式以实现更多的功能。由于该实验为四个死循环任务的执行和切换，

因此程序将持续运行在保护模式，实际上未执行返回实模式并退出的操作。工作模式的切换流程如图 3.3-1 所示：

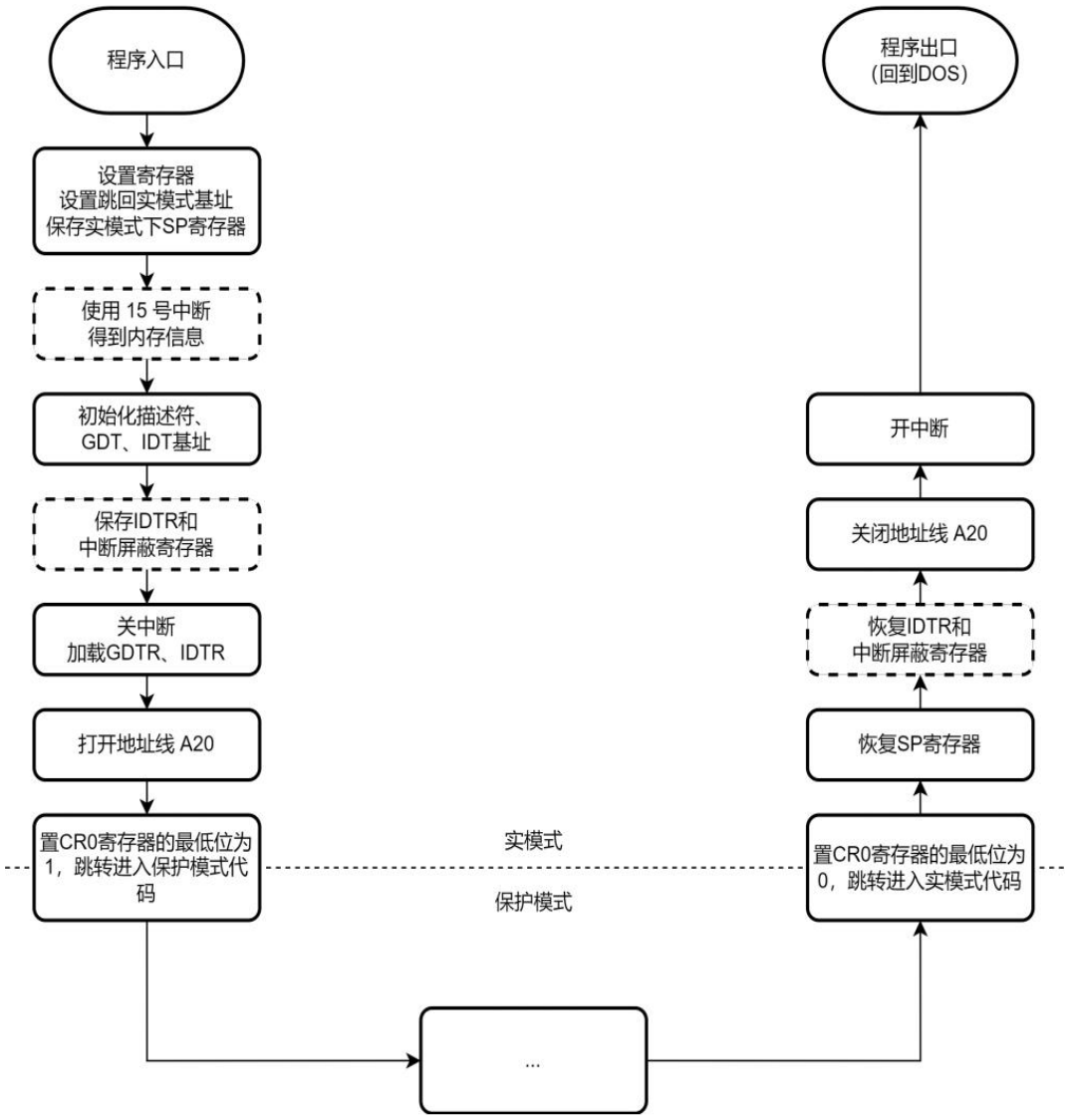


图 3. 3-1 实模式与保护模式的切换

3.4 保护模式初始化

保护模式下 CPU 拥有 32 位的寄存器和 32 位的地址总线，有 4GB 的寻址能力，且借助分页机制可以访问全部的物理内存，并且具备内存保护机制。从实模式通过 jmp 跳转到 32 位代码段进入到保护模式后，还需要完成初始化工作才能使之后的工作顺利进行，此时各段寄存器仍为实模式下的值，需要加载各段选择子来完成各段的初始化，并初始化时钟和中断芯片等等。

3.5 启动分页机制

为实现保护模式下的任务切换，需要开启分页机制，使得相同的线性地址可以通过切换页目录实现到不同物理地址的映射。两级分页机制寻址如下图所示：

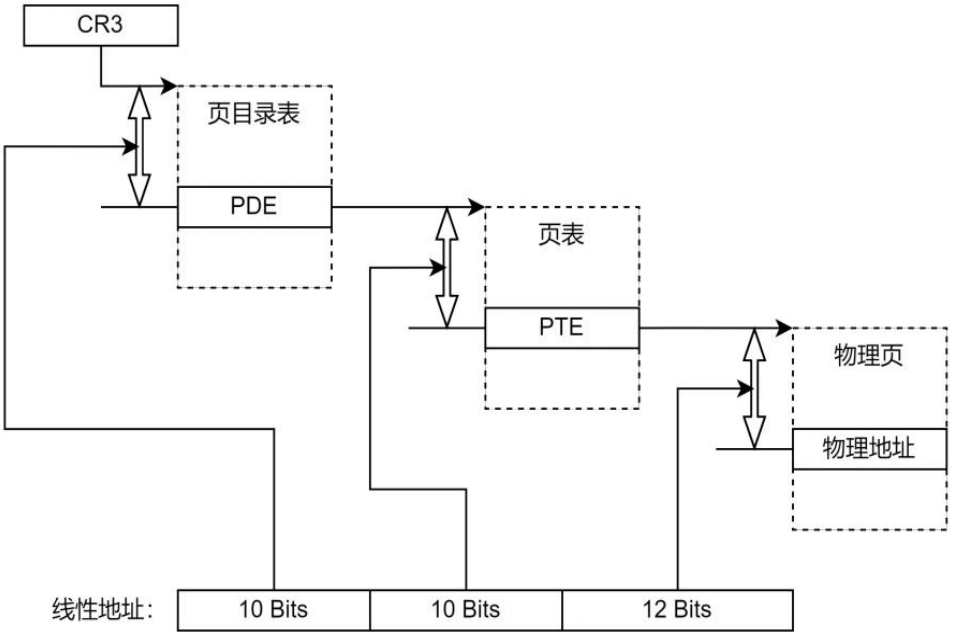


图 3.5-1 分页机制下寻址

在程序中我分别为四个任务定义了四个页目录，第一个页目录是默认的映射关系，其余页目录改变了映射关系，当页目录切换时，同一个线性地址将对应到不同的物理地址。进行转换时，先是从由寄存器 cr3 指定的页目录中根据线性地址的高 10 位得到页表地址，然后在页表中根据线性地址的第 12 到 21 位得到物理页首地址，将这个首地址加上线性地址低 12 位便得到了物理地址。

页目录和页表基址实在数据结构定义模块中完成了，进入保护模式之后，完成的是页目录的初始化以及正式开启分页机制。具体来说，首先需要根据内存大小计算应初始化多少页目录和页表；然后初始化页目录，将选择子存入 es，页目录基址偏移存入 edi；再初始化所有页表；最后需要加载页目录基址到 CR3，加载 TSS 到任务寄存器 TR 中，打开 CR0 的分页标志位 PG 后即开启分页机制。

3.6 时钟中断与任务调度

本模块是整个课设的核心，也是代码编写的重点。任务切换涉及到前面提到的 LDT、TSS、中断门等原理，要求采用优先数调度算法实现对四个死循环显示字符串任务进行切换调度，涉及中断处理、时钟中断、调度算法的设计。

1）时钟中断：中断是一种使 CPU 中止正在执行的程序而转去处理特殊事件的操作。而时钟中断是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。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，通过编写 IDT 中的 20h 号中断处理即可实现时钟中断的修改。而通过修改 8253 芯片的 Counter 0 则可以改变时钟频率。

2）优先数调度算法：调度算法是在多任务情况下决定执行哪个任务的一组规则。在本实验中，采用优先数进程调度策略，选择优先级最高的任务进行调度，直至其时间片用完后，再切换下一个任务。当四个任务都完成后，重置任务的时间片，开始新一轮的执行调度。由此循环反复。

为了实现该算法，我定义了两个数组分别存储任务的优先级和剩余时间片。在任务开始时根据优先级数值初始化时间片，每一次时钟中断发生时，首先判断当前执行的任务是否已经完成（即剩余时间片是否为 0）：若未完成当前任务，则无需切换，继续执行，同时时间片减一；若已完成，则继续判断其他任务是否完成（遍历时间片数组判断是否均为 0），若所有任务都已完成，则重新根据优先数初始化时间片并开始新一轮调度，若有任务未完成，则从中选择时间片最大（即优先级最高）的任务切换执行。如此反复循环，逻辑如下图所示：

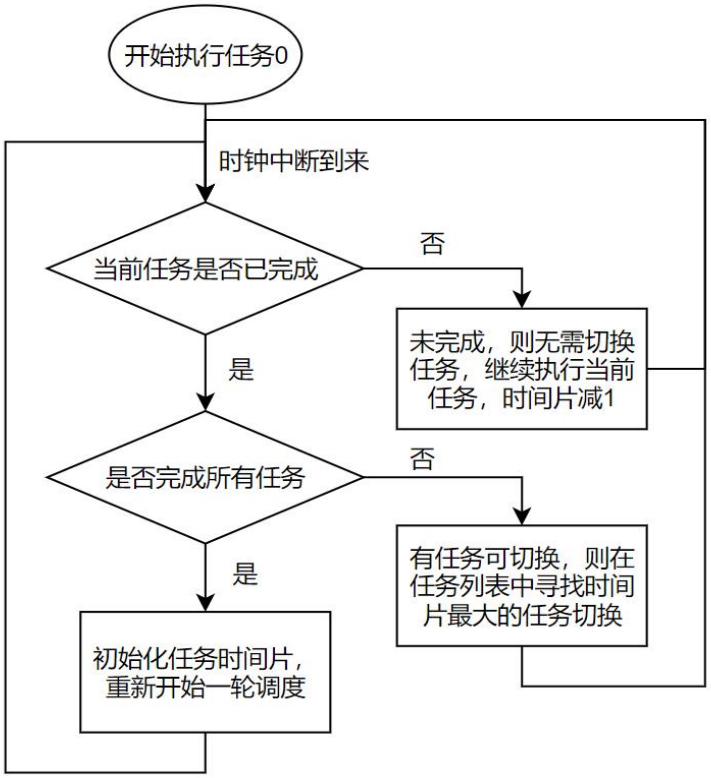


图 3.6-1 优先数任务调度算法

4 实验程序的难点或核心技术分析

4.1 实验流程总结

本实验我完成了任务 4 的部分，即实现保护模式下的优先数调度算法循环显示不同字符串。实验流程具体分为以下几个步骤：

1. 安装配置 bochs 虚拟机
2. 定义并初始化段和描述符表
3. 定义 16 位各功能段
4. 定义 32 位功能段，包括各任务的 LDT 段
5. 进入保护模式
6. 保护模式初始化
7. 初始化任务页目录，开启分页机制
8. 编写时钟中断的任务调度算法
9. 执行任务并利用时钟中断切换

第 1 步参考实验指导没有太大问题；第 2-6 步阅读并理解《于渊：一个操作系统的实现》第 3 章，在示例代码 pmtest9.asm 的基础上编写；第 2 步，第 7-9 步是本实验的重点编写内容。

4.2 实验难点

结合程序设计思路和实验过程分析，可得出本实验的难点主要如下：

1. 进入保护模式

保护模式下的寻址方式与实模式不同，需要使用选择子寻址。实模式下是 16 位代码段，保护模式则是 32 位。进入保护模式分为数据结构的定义、实模式下的准备和进入后初始化三步。这部分主要是阅读理解示例代码并修改。

1）数据结构定义

保护模式依赖 GDT、LDT、selector、Gate、TSS 这几个数据结构，由于描述符、选择子的定义和初始化操作多且重复，故封装成宏汇编以简化代码编写，同时利于阅读和排错。参考书的示例已对数据结构进行宏定义，此处仅展示我自己封装的初始化宏，如下所示：

```
91   ; 物理地址被分成三部分赋值给描述符
92   ; usage: InitDescBase LABEL, LABEL_DESC
93   %macro InitDescBase 2
94       xor eax, eax
95       mov ax, cs
96       shl eax, 4
97       add eax, %1
98       mov word [%2 + 2], ax
99       shr eax, 16
100      mov byte [%2 + 4], al
101      mov byte [%2 + 7], ah
102  %endmacro
```

对于每个任务，需要分别为其定义任务进程 LDT，代码段，数据段，ring0 堆栈段，ring3 堆栈段，任务状态段 TSS。其中 TSS 用于特权级切换下的任务执行，由于 4 个任务一直在特权级下死循环显示字符串，故任务之间的切换无需加载 TSS，但在首次执行任务前需要从用户态跳转到内核态，故必须手动加载 TSS。

```
106  ; 初始化任务描述符，包含LDT，代码段，数据段，堆栈段，TSS
107  ; usage: InitTaskDescBase tasknumber
108  %macro InitTaskDescBase 1
109      InitDescBase LABEL_TASK%1_LDT, LABEL_TASK%1_DESC_LDT        ; 任务进程LDT
110      InitDescBase LABEL_TASK%1_CODE, LABEL_TASK%1_DESC_CODE      ; 代码段
111      InitDescBase LABEL_TASK%1_DATA, LABEL_TASK%1_DESC_DATA      ; 数据段
112      InitDescBase LABEL_TASK%1_STACK0, LABEL_TASK%1_DESC_STACK0  ; 内核堆栈段
113      InitDescBase LABEL_TASK%1_STACK3, LABEL_TASK%1_DESC_STACK3  ; 程序堆栈段
114      InitDescBase LABEL_TASK%1_TSS, LABEL_TASK%1_DESC_TSS        ; 任务状态段TSS
115  %endmacro
```

在每个任务的 LDT 段中，需要存放任务的代码段、数据段、堆栈段等段的描述符，并且需要定义各段对应的选择子。

```
119  ; 定义LDT段
120  ; usage: DefineLDT num
121  %macro DefineLDT 1
122  [SECTION .ldt%1]
123  ALIGN   32
124  LABEL_TASK%1_LDT:
125  ; 描述符                              段基址,       段界限       ,    属性
126  LABEL_TASK%1_DESC_DATA: Descriptor      0,  Task%1DataLen - 1,   DA_DRWA + DA_DPL3
127  LABEL_TASK%1_DESC_CODE: Descriptor      0,  Task%1CodeLen - 1,   DA_C + DA_32 + DA_DPL3
128  LABEL_TASK%1_DESC_STACK0: Descriptor    0,  TopOfTask%1Stack0,   DA_DRWA + DA_32
129  LABEL_TASK%1_DESC_STACK3: Descriptor    0,  TopOfTask%1Stack3,   DA_DRWA + DA_32 + DA_DPL3
130
131  TASK%1LDTLen            equ $ - LABEL_TASK%1_LDT
132  ; LDT 选择子
133  SelectorTask%1Data      equ LABEL_TASK%1_DESC_DATA - LABEL_TASK%1_LDT + SA_TIL + SA_RPL3
134  SelectorTask%1Code      equ LABEL_TASK%1_DESC_CODE - LABEL_TASK%1_LDT + SA_TIL + SA_RPL3
135  SelectorTask%1Stack0    equ LABEL_TASK%1_DESC_STACK0 - LABEL_TASK%1_LDT + SA_TIL
136  SelectorTask%1Stack3    equ LABEL_TASK%1_DESC_STACK3 - LABEL_TASK%1_LDT + SA_TIL + SA_RPL3
137  ; END of [SECTION .ldt%1]
138  %endmacro
```

2）实模式下的准备

计算机刚启动时处于实模式下，需要在 16 位代码段中对 GDT、各段描述符以及任务描述符进行初始化，也就是调用上一步的宏进行操作，并将初始化后的

GDT、IDT 载入对应的寄存器，然后打开地址线 A20，并设置 CR0 寄存器的 PE 位为 1，将 32 位代码段选择子装入 CS 寄存器即可跳转进入保护模式。

3）保护模式初始化

cpu 进入保护模式后寻址方式改变，但各寄存器的值未更改，故需要加载选择子初始化各段寄存器。

2. 分页机制

为了实现任务切换，需要借助分页机制以完成相同线性地址到不同物理地址的映射。本实验要求完成四个任务的切换，故需要设计四套页表并初始化。

```
5   ; 4个任务页目录地址------------------------------------------------------------------------
6   PageDirBase0        equ 200000h ; 页目录开始地址:   2M
7   PageTblBase0        equ 201000h ; 页表开始地址:     2M +  4K
8   PageDirBase1        equ 210000h ; 页目录开始地址:   2M + 64K
9   PageTblBase1        equ 211000h ; 页表开始地址:     2M + 64K + 4K
10  PageDirBase2        equ 220000h ; 页目录开始地址:   2M + 128K
11  PageTblBase2        equ 221000h ; 页表开始地址:     2M + 128K + 4K
12  PageDirBase3        equ 230000h ; 页目录开始地址:   2M + 192K
13  PageTblBase3        equ 231000h ; 页表开始地址:     2M + 192K + 4K
```

需要注意，开始执行任务前，在加载了页目录基址到 CR3 寄存器后还需要手动加载任务 0 的 TSS 到任务寄存器 TR 中，以实现特权级的转换，然后再打开 CR0 寄存器的分页标志位 PG，便能真正开启分页机制。通过切换页目录，可以使得同一个线性地址映射到不同的物理地址，由此实现对不同任务程序的执行。

```
340     ; 开启分页机制
341     mov eax, PageDirBase0   ; ┬ 加载 CR3
342     mov cr3, eax            ; ┘
343     mov ax, SelectorTSS0    ; ┬ 加载 TSS 到任务寄存器TR中
344     ltr ax                  ; ┘
345     mov eax, cr0            ; ┐
346     or  eax, 80000000h      ; ├ 打开分页,标志位PG设为1
347     mov cr0, eax            ; │
348     jmp short .3            ; ┘
349 .3:
350     nop
351     ; 初始化完成，分页机制启动完毕
```

3. 时钟中断机制

本实验需要借助时钟中断机制定期发出中断请求来实现任务调度。具体来说，不仅要设计一个中断处理程序让 IDT 表中 20h 号中断的门选择子指向它，还要设置 IMR，并且设置 IF 位。设置 IMR 可以通过写 OCW2 来完成，而设置 IF 可以通过指令 sti 来完成。且可以通过修改 8253 芯片的 Counter 0 改变时钟频率，使得各任务的显示时间发生改变（也可以通过增加执行的时间片来实现）。

4. 任务调度算法

该部分是本实验的最重点。任务调度是为了让四个任务能够交替显示字符串，采用时间分片的机制来解决单一进程占用 CPU 的问题，让每个任务根据优先级调度算法轮流占用 CPU。需要借助前三项工作来实现，即保护模式下开启分页机制并启用时钟中断。

1） 任务定义

我首先分别定义了四个任务需要显示的字符串和显示参数。对于每个任务，还需要定义 LDT，代码段，数据段，内核堆栈段，用户堆栈段和任务状态段。

任务代码段中将任务字符串逐个字符赋值给 al 寄存器并显示。

```
164   LABEL_TASK%1_CODE:
165   ; 遍历显示szTask%1Message
166       xor     ecx, ecx
167       mov     ah, %3
168   .outputLoop:
169       mov     al, [szTask%1Message + ecx]
170       mov     [gs:((80 * %2 + ecx) * 2)], ax
171       inc     ecx
172       ; 判断是否到达字符串末尾
173       cmp     al, 0
174       jne     .outputLoop
175       jmp     LABEL_TASK%1_CODE
```

2） 任务切换

我封装了一个 SwitchTask 宏用于切换任务。在这个宏函数中，我采用手动加载相关数据的方式进行任务切换。首先要加载目标任务的 LDT 选择子，然后将页目录基址加载到 CR3 中以实现页目录切换，并加载 ds、ss、esp、cs 等寄存器切换任务段，最后使用 iretd 中断返回，任务切换结束。

```
325   ; usage: SwitchTask num
326   %macro SwitchTask 1
327       mov     ax, SelectorLDT%1   ; ┓ 加载 LDT
328       lldt    ax                  ; ┛
329       mov     eax, PageDirBase%1  ; ┓ 加载 CR3
330       mov     cr3, eax            ; ┛
331       ; 切换至任务%1，加载各寄存器
332       mov     eax, SelectorTask%1Data
333       mov     ds, eax
334       push    SelectorTask%1Stack3; SS
335       push    TopOfTask%1Stack3   ; ESP
336       pushfd                      ; EFLAGS
337       pop     eax                 ; ┓
338       or      eax, 0x200          ; ┣ 将 EFLAGS 中的 IF 位置 1，即开启中断
339       push    eax                 ; ┛
340       push    SelectorTask%1Code  ; CS
341       push    0                   ; EIP
342       iretd                       ; 中断返回
343   %endmacro
```

3） 优先数调度算法

在本实验中，采用优先数进程调度策略，选择优先级最高的任务进行调度，直至其时间片用完后，再切换下一个任务。当四个任务都完成后，重置任务的时间片，开始新一轮的执行调度。每个时间片 CPU 都会发生一次时钟中断，并调用时钟中断处理程序，即我编写的任务调度程序对任务进行处理。

为实现该算法，我定义了两个数组 TaskPriority 和 ExecutionTicks 分别存储任务的优先级和剩余时间片。在任务开始时根据优先级数值初始化时间片，每次时钟中断发生时，先判断当前任务是否已完成（即剩余时间片是否为 0）：若未完成，则无需切换，继续执行，时间片减一。

```
401     ; 判断ExecutionTick是否为0，如果不为0，则说明当前任务未执行完，无需进行任务切换
402     mov     edx, dword [RunningTask]
403     mov     ecx, dword [ExecutionTicks+edx*4]
404     test    ecx, ecx
405     jnz     .subTicks   ; 无需进行任务切换，跳到subTicks，继续执行当前任务且时间片-1
406     ; 否则，当前任务已经执行完成。
407     ; 继续判断任务是否已经全部完成，是则重新赋值
408     mov     ebx, edx
409     mov     eax, dword [ExecutionTicks]
410     or      eax, dword [ExecutionTicks + 4]
411     or      eax, dword [ExecutionTicks + 8]
412     or      eax, dword [ExecutionTicks + 12]
413     jz      .initTask   ; 任务全部完成，重新赋值
```

若已完成，则继续判断其他任务是否完成（遍历时间片数组判断是否均为 0）。若有任务未完成，则从中选择时间片最大（即优先级最高）的任务切换执行。通过循环遍历 ExecutionTicks 数组来找最大值。

```
414     ; 否则没有全部完成，进行任务切换
415 .goToNext:  ; 选择下一个任务
416     xor     esi, esi
417     xor     eax, eax
418     xor     ecx, ecx
419 .getMaxLoop:  ; 获取Ticks最大的任务
420     cmp     dword [ExecutionTicks+eax*4], ecx
421     jle     .notMax
422     mov     ecx, dword [TaskPriority+eax*4]
423     mov     ebx, eax
424     mov     esi, 1
425 .notMax:
426     add     eax, 1
427     cmp     eax, 4
428     jnz     .getMaxLoop     ; 循环获取Ticks最大的任务
```

找到最大值后，将其对应的任务编号存入 edx 寄存器和 RunningTask 变量中，然后调用 SwitchTasks 宏切换到目标任务，接着执行任务，时间片-1。

```
434     SwitchTasks     ; 切换到目标任务
435 .subTicks:  ; 继续执行当前任务，时间片-1
436     sub     dword [ExecutionTicks+edx*4], 1
437     jmp  .exit
```

如果所有任务都完成了，则重新根据优先级初始化各任务的时间片，准备开启新的一轮调度。

```
438     .initTask:  ; 全部任务均已结束，重新赋值
439         xor     ecx, ecx
440     .setLoop:
441         mov     eax, dword [TaskPriority + ecx*4]
442         mov     dword [ExecutionTicks + ecx*4], eax
443         inc     ecx
444         cmp     ecx, 4
445         jne     .setLoop
446         xor     ecx, ecx
447         jmp     .goToNext   ; 赋值后按照优先级选择任务
448     .exit:
449         popad
450         pop     ds
451         iretd
```

到此，一次时钟中断完成。总的处理逻辑如下流程图所示：

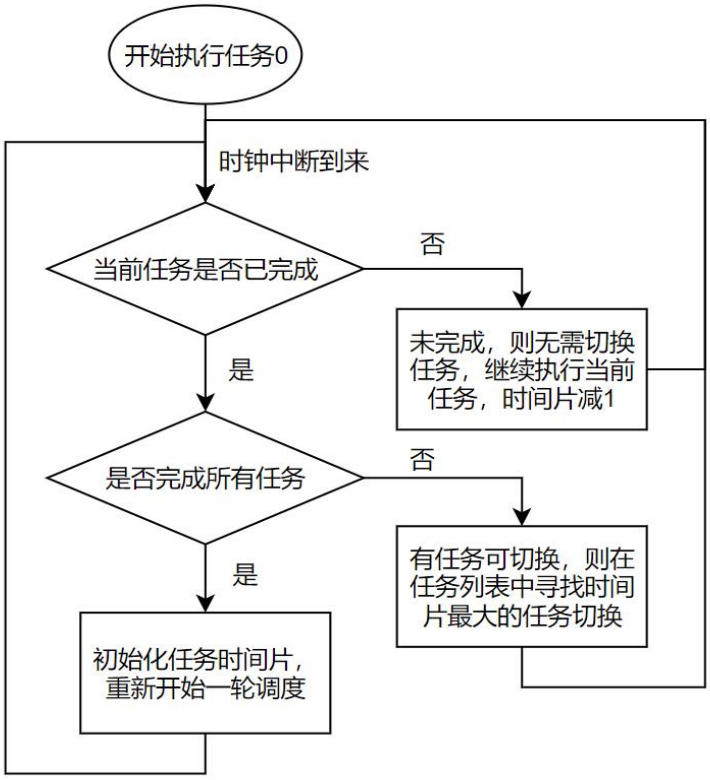


图 4-1 优先数任务调度算法

5 开发和运行环境的配置

5.1 开发和运行环境

主机：Windows 11

集成开发环境：Visual Studio Code1.87.0 (user setup)

实验环境：

虚拟机：

VMWare Workstation 17 Pro 17.0.0 build-20800274

Ubuntu 20.04.6 LTS

Bochs x86-64 emulator

工具：NASM version 2.14.02

bochs 的配置文件如下所示：

```
megs: 32
romimage: file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/share/bochs/VGABIOS-lgpl-latest
floppya: 1_44=freedos.img, status=inserted
floppyb: 1_44=pmtest.img, status=inserted
boot: a
mouse: enabled=0
```

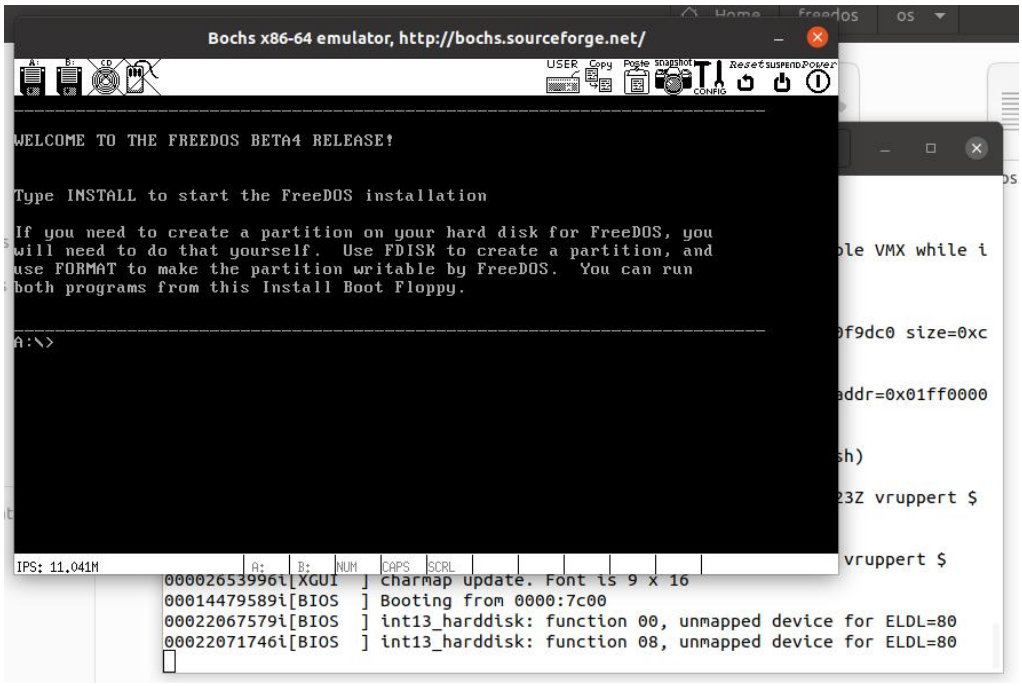
5.2 程序更新、运行、调试

配置好 bochs 之后，为了更方便地进行程序的更新和测试，我编写了 makefile 文件用于命令管理，直接在项目目录终端中执行 make 可一键完成编译汇编代码、挂载 b 盘、更新 com 程序、启动 bochs 等操作，makefile 内容如下：

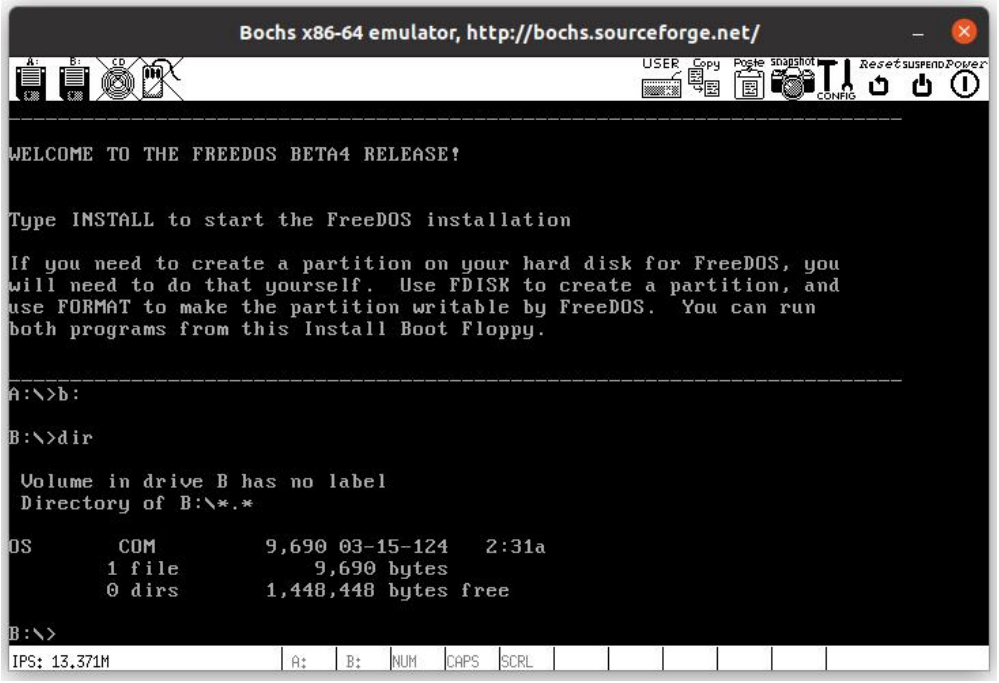
```
# 在make之前目录中需要准备好freedos.img和格式化的空白盘pmtest.img
all: com image bochs
# 编译汇编代码
com:
	nasm os.asm -o os.com
# 挂载b盘并写入文件
image:
	- sudo rm -rf /mnt/floppyb/*
	mkdir -p /mnt/floppyb
	- sudo umount /mnt/floppyb
	sudo mount -o loop pmtest.img /mnt/floppyb
	sudo cp os.com /mnt/floppyb/os.com
# 启动bochs
bochs:
	bochs -f bochsrc.txt
# 清空
clean:
	sudo rm -rf /mnt/floppy/*
	- sudo umount /mnt/floppyb
.PHONY:
	all
```

6 运行和测试过程

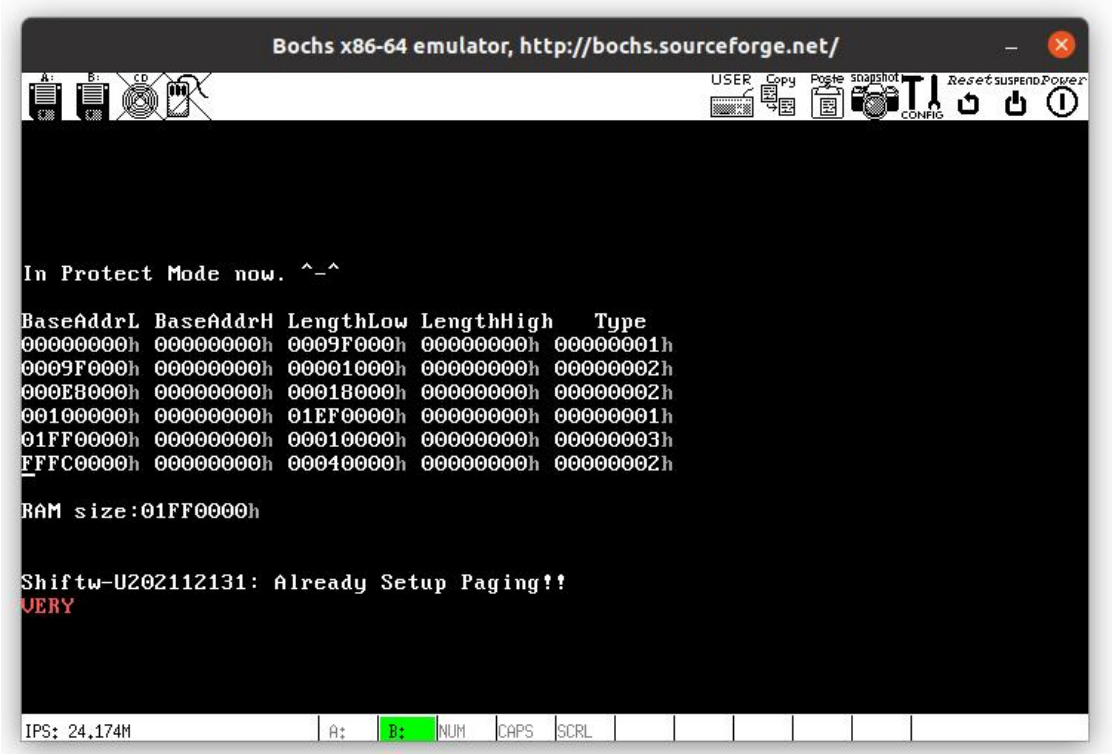
在虚拟机中打开终端进入项目路径，执行 make 命令后编译程序并写入软盘 b，bochs 虚拟机启动并弹出新窗口，在原来的终端中输入 c 回车继续执行，可以观察到 bochs 虚拟机窗口初始化完成，显示已进入 freedos。

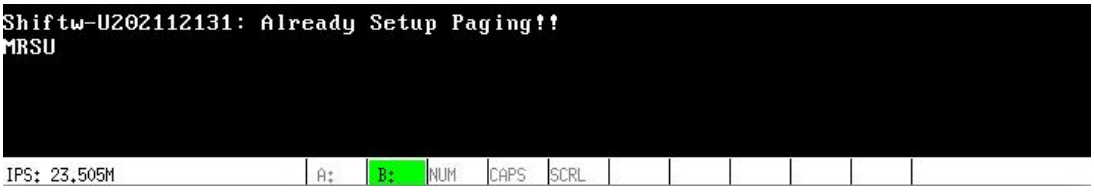


此时 CPU 处在系统的 a 盘中，在光标处输入命令切换到 b 盘并查看其下内容，可见编译好的 com 程序已在 b 盘中。



输入文件名 os.com 回车即可运行程序，可观察到清屏操作完成并打印出内存信息和提示信息，表明已进入保护模式并开启分页机制。四个字符串在窗口处固定位置交替显示，且每个字符串的显示时间与优先级有关，VERY 显示时间最长，LOVE 其次，再到 HUST，MRSU 最短。





7 实验心得和建议

在本次实验中，我实现了保护模式下借助分页机制和时钟中断完成的优先数任务调度。其中涉及了实模式切换到保护模式、保护模式的段页式机制、时钟中断和任务调度算法等等知识，是对理论课的深入和扩展实践，让学生从代码实践层面理解一个操作系统的实现。前期时间主要用在阅读和理解《于渊：一个操作系统的实现》书籍和示例代码上，最终的项目在此基础上完善了时钟中断的优先数调度算法。

一开始我没有很好地将本次实验和上学期学习的理论知识结合起来理解，导致虽然各个部分都大致掌握，但整合在一起去实现的时候却是困难的。比如之前对于保护模式只有一个概念性的理解，至于保护模式的实现细节，是非常模糊的。不过通过这次实验，基本上清晰了进入保护模式的操作。虽然是参考了书本代码，但实现了那一个伟大的 jmp 指令还是很激动的。

虽然指导书对代码和原理的解释都很清晰，但是要将保护模式的多个知识点串联在一起还是有一定难度，并且在代码的实现细节上也仍会遇到一些困难和不解，下面一一列出：

（1） 字符串只显示一次就结束：

原因是重复调用时钟中断需要发送 EOI 给 8259A，此操作意义是发送本次中断的结束信号并自动产生下一次时间中断。如果遗漏了该操作则 8259A 芯片会一直等待中断结束信号，不结束当前中断。所以补上该操作即可解决问题。

（2） 初始化描述符、选择子和各段时，代码重复且冗余，不便于修改和排错：

示例代码中定义了各数据段的宏，由此我得到思路将重复的代码封装成宏函数使用，大大减少了代码量，而且整个程序阅读起来也更清晰，编写时也不容易出错。

（3） 寄存器无法作为宏函数参数

在编写任务切换宏时，因目标切换任务编号存在寄存器中，原本的想法是直接将 edx 的值作为参数，简化代码执行。但 NASM 不支持这个操作，改用多次判断分支直接用立即数传参。

（4） 使用 dup 定义多个相同字节报错：

原因是 NASM 相较于 MASM 取消了 dup 的使用，将 db 512 dup 0 改为 times 512 db 0 即可。

（5） 任务无法正常进行，字符串不显示或无法切换：

这个问题是最大的一个问题，原因有多个，具体如下：

一是没有初始化所有的页目录和页表项，只初始化了第一套页表，导致之后任务没办法正常切换。

二是首次执行任务前除了加载页目录基址到 CR3 之外还需要加载 TSS。因为在首次执行任务的时候实际上有一个特权级转换过程，即从用户态转换到内核态，必须加载 TSS 任务状态段进行切换。但是之后的任务切换不会发生特权级转换，因为都是在内核态下执行的，所以不需要加载 TSS。

三是各段的描述符中 DPL 设置不正确。DPL 规定访问该段的权限级别，每个段的 DPL 固定。当进程访问一个段时，需要进程特权级检查，一般要求 DPL >= max {CPL, RPL}，即级别至少要相同才能通过检查。

四是中断使用 int 指令执行，iretd 返回。使用 iretd 中断返回时要压入标志寄存器 EFLAGS，并将其中的 IF 标志位设为 1，即开启中断。即 irted 前的栈从底到顶应为 SS、ESP、EFLAGS、CS、EIP。

（6） 打印信息直接显示在固定位置，背景信息未清空影响观察：

在打印信息之前进行屏幕清空操作，即在整个屏幕范围内打印黑底黑字覆盖掉之前的文字。

（7）代码更新和程序运行频繁，执行多条命令操作麻烦：

编写一个 makefile 文件用于命令管理，将编译、挂载软盘、启动虚拟机等指令整合，每次只需要执行 make 即可。

总之，非常感谢老师们精心设计的这门课设，虽然给人的感觉更像是实验课，因为大部分代码参考书中均已给出。自主阅读参考书并研读代码的方式使得我在编写代码的时候有着较为清晰的思路，对操作系统保护模式的实现细节也有了更深的理解，总之收获颇丰。

最后是一点小建议：虽然课程参考书已经解释得很详尽了，但仍觉得和之前的理论课脱节有些大。建议加强理论课学习中的相关部分。课设开始后一周多的

时间都是在学习和理解示例代码，虽然最后编写的代码主要就时钟中断处理的部分，但对于其它部分的理解和使用反而是最耗时的，做到最后仍有小部分底层操作难以理解到位。且课程测试有一定难度，考察得太细节了，有些点可能在实验过程中没有遇到太多困难，印象不深。更建议老师们在课程开始的时候给出一些关键点提示，给同学们提供更多角度的思考。

8 学习和编程实现参考网址

（1）《操作系统原理》教材

（2）《于渊：一个操作系统的实现》

（3）课件

（4）https://www.cnblogs.com/alwaysking/p/12287116.html

（5）https://www.cnblogs.com/chenwb89/p/operating_system_004.html

（6）https://blog.csdn.net/LinuxArmbiggod/article/details/122375763

（7）https://www.anquanke.com/post/id/259368

（8）https://blog.csdn.net/q1007729991/article/details/52650822