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：

学　　号：

姓　　名：

2024 年 04 月 07 日

目　录

1 课设目的

理解保护模式基本工作原理：保护模式是一种计算机处理器的工作模式，通过内存分段和分页、特权级别、中断异常处理以及访问控制等机制，实现对操作系统和应用程序的保护和隔离，确保系统的安全性、稳定性和可靠性。

理解保护模式地址映射机制：指通过分段和分页技术，将程序所使用的虚拟地址映射到物理内存上，实现内存的管理和保护。分段将内存划分为不同的段，并赋予每个段不同的权限，而分页将内存划分为固定大小的页面，并通过页表将虚拟地址映射到物理地址，实现了内存的虚拟化和隔离，同时提供了更细化的内存访问控制和保护，确保了系统的安全性和稳定性。

段机制和页机制：段机制将内存划分为不同的段，每个段都有自己的起始地址和长度，并且可以被赋予不同的权限，从而实现了对内存的逻辑划分和访问控制；而页机制将内存划分为固定大小的页面，通过页表将虚拟地址映射到物理地址，实现了内存的虚拟化和隔离，同时提供了更细粒度的内存管理和访问控制，使得操作系统可以更有效地管理内存资源，确保系统的安全性和稳定性。

理解任务/进程的概念和切换过程：任务或进程是操作系统中的基本执行单元，它包含了程序的代码、数据和执行状态。操作系统需要有效地管理和调度多个任务或进程的执行，以便在有限的资源下实现高效的并发处理。在多任务环境下，操作系统通过上下文切换来实现任务或进程之间的快速切换，其中保存当前任务或进程的执行状态，加载下一个任务或进程的状态，并确保执行的连续性，以实现多任务之间的无缝切换和并发执行。

理解优先数进程调度原理：优先数进程调度原理是一种操作系统调度算法，它根据每个进程的优先级来确定下一步执行的进程。优先数越高的进程会被优先选择执行，以保证高优先级任务的及时响应，低优先级的进程则暂时等待调度。调度器按照优先级高低依次选择进程执行，每个进程运行一定时间后，调度器重新安排下一个进程，直至所有进程都运行完成其对应的时间片。

编程技能培养：首先学习了裸机编程的基础知识，包括汇编语言和底层硬件操作。然后，我深入了解了保护模式的初始化过程，建立全局描述符表（GDT）、

页表等操作。接着，我实现了多任务系统，包括任务的创建、调度和切换。最后，我编写了优先数调度算法，确保高优先级任务得到优先执行。这个过程不仅提升了编程技能，还加深了对操作系统内核原理的理解。

掌握保护模式的初始化：在保护模式初始化过程中，操作系统需完成诸如建立全局描述符表（GDT）、加载各种系统数据结构（如 GDT 表、IDT 表）、建立页表、设置中断向量表、加载各类寄存器等操作。这些步骤的完成使得处理器能够进入保护模式，并且为操作系统提供了必要的内存保护和多任务管理功能，从而启动了操作系统的全面运行。

掌握段机制的实现：在段机制下，内存被划分为多个段，每个段有自己的起始地址和长度，并且可以被赋予不同的访问权限。操作系统通过段描述符表（如 GDT）来管理这些段，每个段描述符包含了段的起始地址、长度和权限等信息。在程序执行时，CPU 根据段选择子来选择段描述符，从而确定要访问的段，并根据段描述符的权限检查访问合法性。通过理解段机制的实现，可以深入掌握操作系统内存管理的工作原理和机制。

掌握页机制的实现：页机制将物理内存划分为固定大小的页面，通过页表将虚拟地址映射到物理地址。操作系统维护页表，其中包含了虚拟地址和对应的物理地址之间的映射关系。当程序访问虚拟地址时，CPU 会通过页表查找对应的物理地址，并进行访问。页机制实现了虚拟内存的概念，使得操作系统可以更灵活地管理内存，实现了内存的隔离和保护，以及对物理内存的高效利用。

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，包含程序的代码、数据和执行状态。任务切换是指操作系统在多任务环境下，通过保存当前任务的状态并加载下一个任务的状态，实现不同任务之间的快速切换，从而实现多任务并发执行的功能。通过理解任务的定义和任务切换的实现机制，可以深入了解操作系统的调度算法和内核设计，以及实现高效的多任务处理。

掌握保护模式下中断程序设计：在保护模式下，中断是处理器在发生特定事件时暂停当前执行的程序并转而执行预定义的中断处理程序。中断程序设计涉及定义和注册中断处理程序、保存和恢复中断上下文、响应中断事件并进行相应处理等关键步骤。通过深入了解中断程序设计，可以掌握操作系统如何管理和响应各种系统事件，实现系统的可靠性和稳定性。

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

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。在保护模式下，CPU 有着巨大的寻址能力，并为强大的 32 位操作系统提供了更好的硬件保障。

在实模式下，16 位的寄存器需要用“段：偏移”这种方法才能达到 1MB 的寻址能力，段值可以看做是地址的一部分，物理地址（Physical Address）=段值（Segment）×16+偏移（Offset）。

保护模式下，拥有 32 位的寄存器和 32 位的地址总线，寻址空间达到 4GB。虽然段值仍然由原来 16 位的 cs、ds 等寄存器表示，但此时它仅仅是一个索引，这个索引指向 GDT 或 LDT 的一个表项描述符，描述符中详细定义了段的起始地址、界限、属性等内容。

从实模式到保护模式的切换过程的设计和实现思路如下：

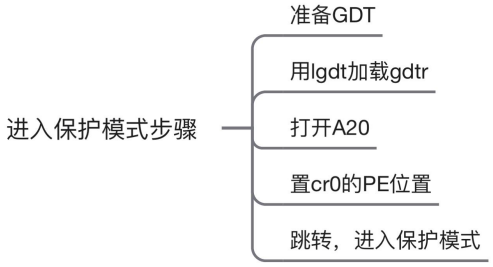

图 3-1 进入保护模式步骤示意图

1）准备 GDT：首先，定义全局描述符表 GDT，包括一个代码段描述符和一个显存描述符，这些描述符是在保护模式下访问内存的必要准备。

2）用 lgdt 加载 gdtr：通过将 GDT 基地址加载到 gdtr 寄存器，即 lgdt [GdtPtr] 指令，将 GDT 表加载到处理器中。

3）打开 A20：打开 A20 地址总线。

4）置 cr0 的 PE 位：将 cr0 寄存器的保护模式 PE 位设置为 1，启用保护模式，通过将 cr0 加载回 cr0 寄存器实现，即 mov cr0, eax 指令。

5）跳转，进入保护模式：使用 jmp dword SelectorCode32:0 指令，跳转到 32 位代码段执行，它包含保护模式所需的指令和数据。在 32 位代码段中，继续进行初始化操作，并执行简单的显示操作，作为成功进入保护模式的标志。

3.2 GDT

全局描述符表 GDT 是存储段描述符的数据结构，用于定义内存段的属性和位置。GDT 包含了一系列描述符，每个描述符描述一个内存段，包括基地址、段限长、访问权限等信息。

段描述符是用于描述内存段属性和位置的数据结构。每个段描述符包含了段的基地址、段的界限、访问权限以及其他一些标志位。示意图表示的是代码段和数据段描述符，描述符的种类还有系统段描述符和门描述符。

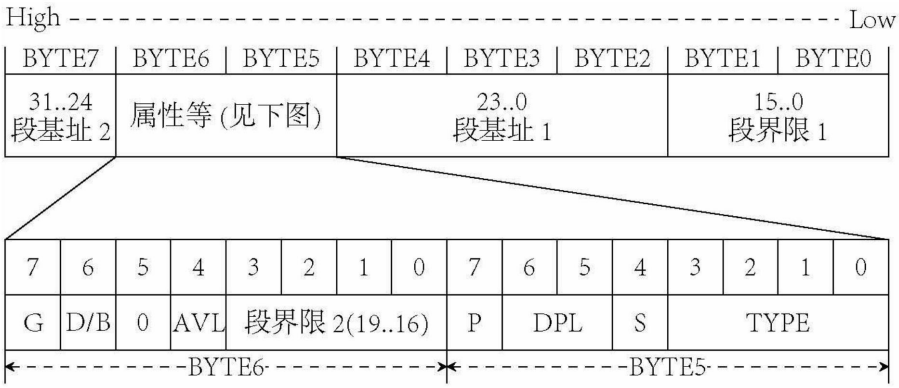

图 3-2 代码段和数据段描述符

描述符各属性如下图：

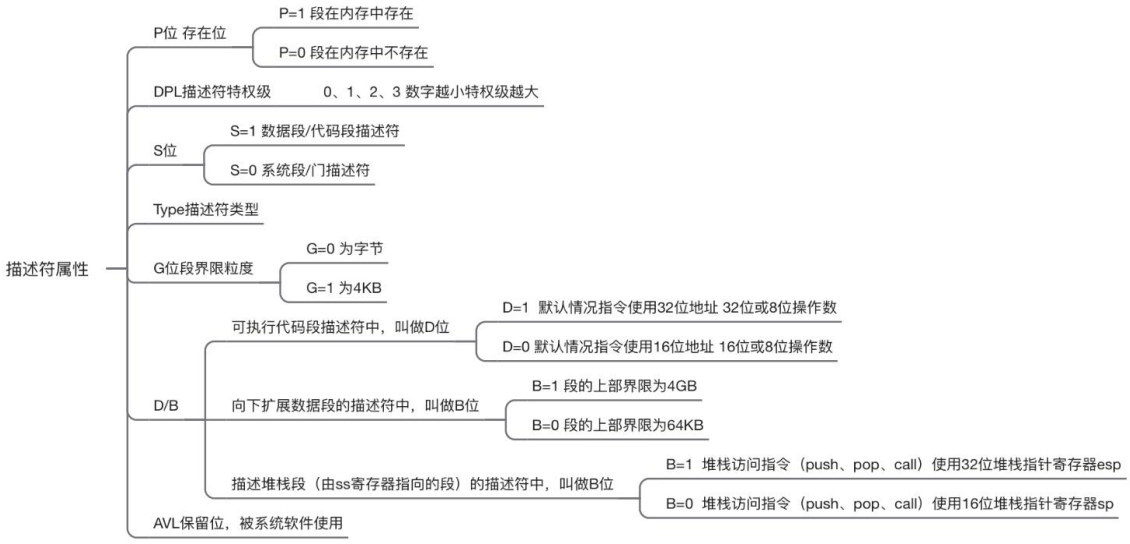

图 3-3 描述符属性示意图

选择子包含段描述符在 GDT 中的索引号和请求者的特权级别信息，用于在保护模式下确定要访问的段描述符，并进行访问权限的检查。其结构包括段索引、

表指示器和请求者特权级别字段。

图 3-4 选择子结构图

GDTR 存储了 GDT 的基地址和界限，用于在保护模式下加载和访问 GDT。

图 3-5 gdtr 结构图

GDT 表的设计与实现思路：

1）定义 GDT 段：在内存中定义 GDT 段，并在其中定义所需的段描述符，包括 32 位代码段、16 位代码段、数据段、堆栈段、任务的 LDT 段以及 TSS 段描述符等。

2）定义 GDT 段描述符的选择子：为每个段描述符定义选择子，用于在程序中标识和访问这些描述符。

3）初始化 GDT 段描述符：在进入保护模式之前，对 GDT 段描述符进行初始化，包括设置每个描述符的基址、长度以及访问权限等属性。

4）加载 GDT 段寄存器：使用 lgdt 指令加载 GDT 段寄存器，将 GDT 的基地址和长度加载到 GDTR 寄存器中，使处理器能够访问和识别 GDT 中的段描述符。

3.3 LDT

LDT 跟 GDT 都是描述符表（Descriptor Table），区别仅仅在于全局（Global）和 局部（Local）的不同。

LDT 的描述符位于 GDT 内，而 LDT 的其他段描述符位于一个单独的描述表内，该局部描述符表通过 LDT 的选择子进行寻址。寻址过程涉及使用 LDT 选择子来寻址 LDT 描述符表，通过段选择子来寻址段的基址。段描述符和选择子的概念与 GDT 相同，但 LDT 寄存器 LDTR 不同于 GDT 寄存器 GDTR，它是一个 16 位的选择子。

LDT 表的设计与实现思路：

1）在全局描述符表 GDT 中定义 LDT 段描述符与选择子：在 GDT 中定义一个 LDT 段描述符，描述 LDT 的基址和长度等属性，并为其分配一个选择子，以便在后续访问时能够唯一标识和定位 LDT。

2）定义 LDT 段：其中包含了任务所需的各种段描述符，如任务段、数据段、堆栈段等，并为每个描述符分配一个局部段选择子，以便在任务切换时能够正确访问这些段。

3）加载 LDT 到 LDTR 寄存器内：在任务切换时，将 LDT 的选择子加载到 LDTR 寄存器中，以便处理器能够根据 LDTR 中的选择子来访问 LDT，从而能够正确寻址任务的私有段。

3.4 TSS

任务状态段（Task State Segment，TSS）用于存储处理器执行任务时的状态信息。TSS 包含了任务的各种状态，如处理器寄存器的内容、任务的特权级别、任务的段选择子等。

TSS 的原理是通过切换不同的 TSS 来实现任务的切换。当处理器执行任务切换时，会加载新的 TSS，从而切换到相应的任务。TSS 中包含了任务的各种状态信息，处理器在加载新的 TSS 时，会根据其中的信息更新处理器的状态，以确保执行新任务时能够恢复到正确的状态。

通过 TSS，处理器可以在任务间进行快速切换，并确保执行任务时的状态正确和一致，TSS 在多任务操作系统中用于实现任务的管理和切换。

TSS 的结构包括几个关键部分：

1）TSS 段描述符：用于描述 TSS 在 GDT 中的位置和属性，包括 TSS 的基址和长度等信息。

2）TSS 中的任务状态：包括处理器寄存器的内容（如 EIP、ESP 等）、任务的特权级别、任务的段选择子等。

3）IO 映射基址：用于指定任务的 I/O 映射位图的地址，用于实现任务级别的 I/O 访问权限控制。

4）重点关注偏移 4 到偏移 27 的 3 个 ss 和 3 个 esp。当发生堆栈切换时，内

层的 ss 和 esp 就是从这里取得的。

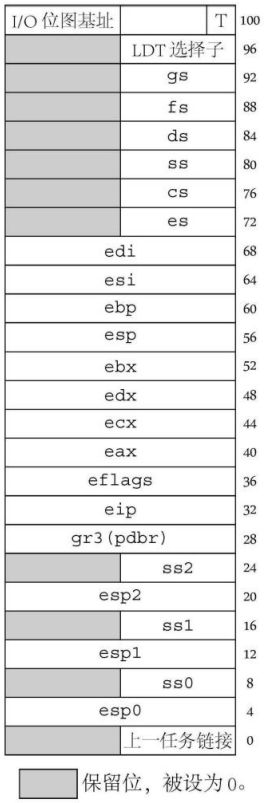

图 3-6 32 位 TSS 结构图

TSS 的设计与实现思路：

1）定义 TSS 段描述符：在全局描述符表 GDT 中定义了一个 TSS 段描述符，指定 TSS 在内存中的位置和属性以及 TSS 的长度。

2）定义 TSS 段：包括任务的各种状态信息，如栈顶指针、堆栈段选择子、代码段选择子、数据段选择子、任务状态等。其中，TopOfStack 指向 0 级堆栈顶部，SelectorStack 为 0 级堆栈选择子。

3）定义 I/O 位图：在 TSS 段中定义 I/O 位图，用于实现任务级别的 I/O 访问权限控制。I/O 位图的基址由 I/O Map Base Address 字段指定，通过设置对应位来控制任务对 I/O 端口的访问权限。

4）加载 TSS 段：在任务切换时，将 TSS 段的选择子加载到特定的寄存器 LDTR 中，以便处理器能够正确寻址和访问 TSS 段。同时，处理器在加载新的 TSS 时，会根据其中的任务状态信息来更新处理器的状态，以确保执行新任务时能够恢复到正确的状态。

3.5 切换特权级任务

特权级是指处理器在执行指令时的权限级别，通常用于区分操作系统内核态（ring0）和用户态（ring3）等不同的权限级别，通过 CPU 的特权级别位 CPL 和段描述符的特权级别 DPL 来实现，以保护系统资源免受未授权访问，并确保系统的稳定性和安全性。本实验中时钟中断处于内核态，用户代码运行于用户态，所以在进行任务调度时需要进行特权级切换操作。

切换特权级任务的设计与实现思路：

1）加载 TSS：将要切换到的任务的 TSS 段选择子加载到 LDTR 寄存器中，以便处理器能够访问和识别任务的 TSS 段。

2）加载 LDT 段描述符：将任务的 LDT 段描述符加载到 LDTR 寄存器中，以便处理器能够访问任务的私有段。

3）修改 ds 寄存器：将数据段选择子加载到 ds 寄存器中，以确保后续指令可以正确访问任务的数据段。

4）进行任务切换：使用 iretd 指令执行任务切换。iretd 指令是一种中断返回指令，与 iret 指令类似，可以在任务切换时更新特权级，使得处理器能够正确恢复到新任务的执行环境中。

3.6 页式存储

在 80386 处理器中，一页是指内存的一个固定大小的块，大小为 4096 字节（4KB）。在未启用分页机制时，逻辑地址直接等同于物理地址，而分段机制将逻辑地址直接映射到物理地址。当启用分页机制时，分段机制首先将逻辑地址转换为线性地址，然后线性地址再通过分页机制转换为物理地址，这样可以更加灵活地管理内存和提供更多的内存隔离和保护。

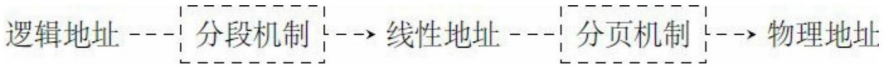

图 3-7 分页开启时的地址转换

PDE（页目录项）和 PTE（页表项）是 x86 架构中用于管理页表的数据结构，其中的各位提供了关于页或页表的重要信息。包括 P（存在位）、R/W（读写位）、U/S（用户/系统位）、PWT（页面写入类型位）、PCD（页面缓存禁用位）、A（访问位）、D（脏位）、PS（页大小位）、PAT（页面属性表选择位）、G（全局位）

等。它们共同构成了页表的核心属性，通过操作这些位可以控制页表的行为，包括页的存在性、读写权限、用户级别权限、页面缓存策略、页面访问记录、页面写入记录、页大小、页面属性等。

cr3（Page-Directory Base Register）是一个控制寄存器，其中的高 20 位存储了页目录表的首地址的高 20 位，而低 12 位为 0，保证了页目录表的 4KB 对齐。它起着指示当前任务页目录表位置的作用，在任务切换时，操作系统会修改 cr3 的值以加载不同的页目录表，从而实现虚拟内存的管理和地址转换。

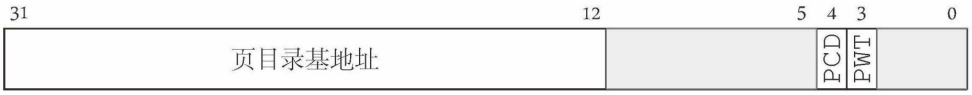

图 3-8 cr3 结构

实现分页机制的设计与实现思路：

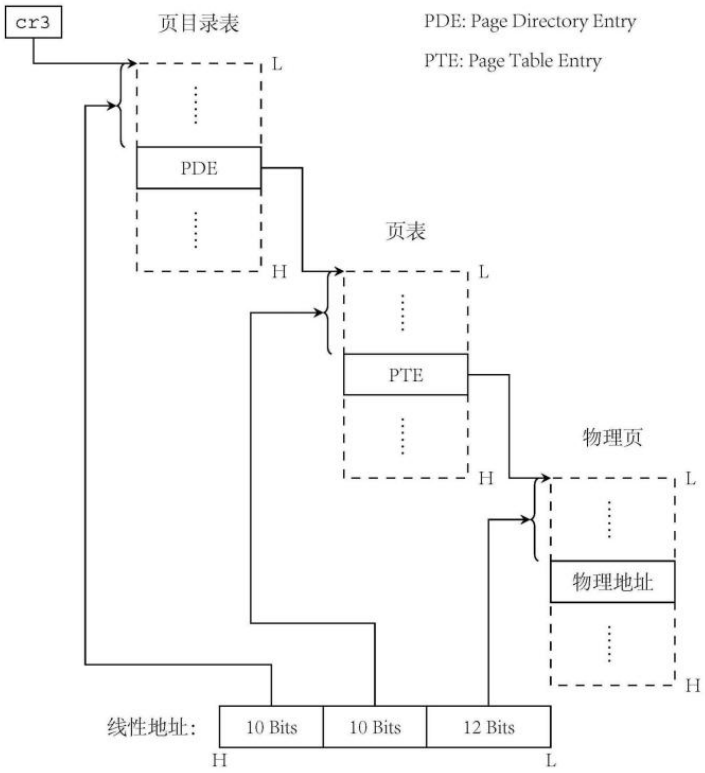

图 3-9 分页机制示意

1）定义页目录和页表结构：采用两级页表结构，其中第一级为页目录，第二级为页表。确定页表的大小和表项格式。

2）初始化页目录和页表：在物理内存中分配空间来存储页目录和页表，初始化页目录，确定每个页目录项对应的页表的物理地址，并初始化页表，确定每

个页表项对应的物理页的物理地址。

3）加载页目录表地址：将页目录表的物理地址存储到控制寄存器 CR3 中，以便处理器能够找到页目录表。

4）启用分页机制：将 CR0 控制寄存器中的分页标志位（PG 位）设置为 1，以启用分页机制，使得处理器开始根据页目录和页表进行地址转换。

5）进行地址转换：当处理器访问内存时，会将线性地址转换为物理地址。处理器会根据线性地址的高位索引页目录表，然后根据页目录项中的页表基址找到相应的页表，再根据线性地址的中低位找到对应的物理页框，最终得到物理地址

3.7 时钟中断和调度算法

时钟中断是由操作系统设置的硬件定时器产生的周期性中断请求，用于通知操作系统进行任务切换。中断处理程序是处理时钟中断的程序，响应中断并保存当前进程的状态，然后将控制权交给调度器。调度算法是一组规则，用于确定下一个要执行的任务。在优先数进程调度算法中，根据任务的优先级来确定执行顺序，优先级高的任务被优先执行，直至其时间片用尽或有更高优先级的任务出现。这些机制共同实现了多任务处理，提高了系统的效率和响应能力。

8259A 可编程中断控制由主片和从片组成，主片负责管理整个系统的中断控制，而从片则协助扩展主片的中断处理能力。它包含了多个中断请求线路的控制和管理逻辑，其中包括中断请求的屏蔽、优先级处理和中断向 CPU 的传递等功能，8259A 的结构如下图：

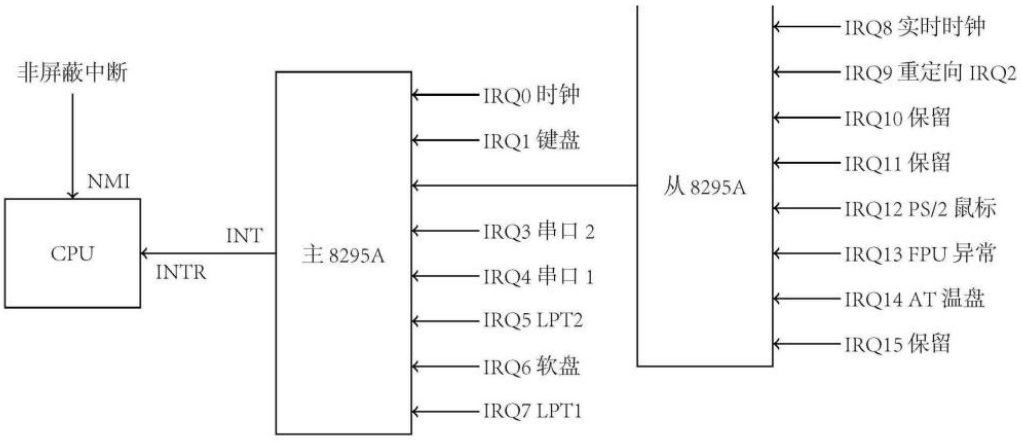

图 3-10 8259A

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，配置其定时产生时钟中断请求的频率和间隔。

2）建立中断描述符表（IDT），为时钟中断分配一个中断向量，并设置对应的描述符和选择子。

3）加载 IDT 表，将其基址和界限加载到 IDTR 寄存器中，使系统能够正确响应中断请求。

4）编写时钟中断处理程序，即中断服务例程（ISR），在其中编写调度算法的逻辑。这包括根据优先级选择要执行的任务，并进行任务切换。

5）在初始化完成后，打开时钟中断响应，使系统能够响应来自 8259A 的时钟中断请求。

6）系统等待时钟中断的到来，当时钟中断触发时，CPU 会自动跳转到时钟中断处理程序，执行调度算法，完成任务的切换和调度。

4 实验程序的难点或核心技术分析

4.1 实验流程

本实验我分成了两部完成，首先完成 task1、task2、task3，实现了程序内存的分布以及根据时钟中断进行任务的切换，最后在 task4 中重点实现了优先数调度算法。

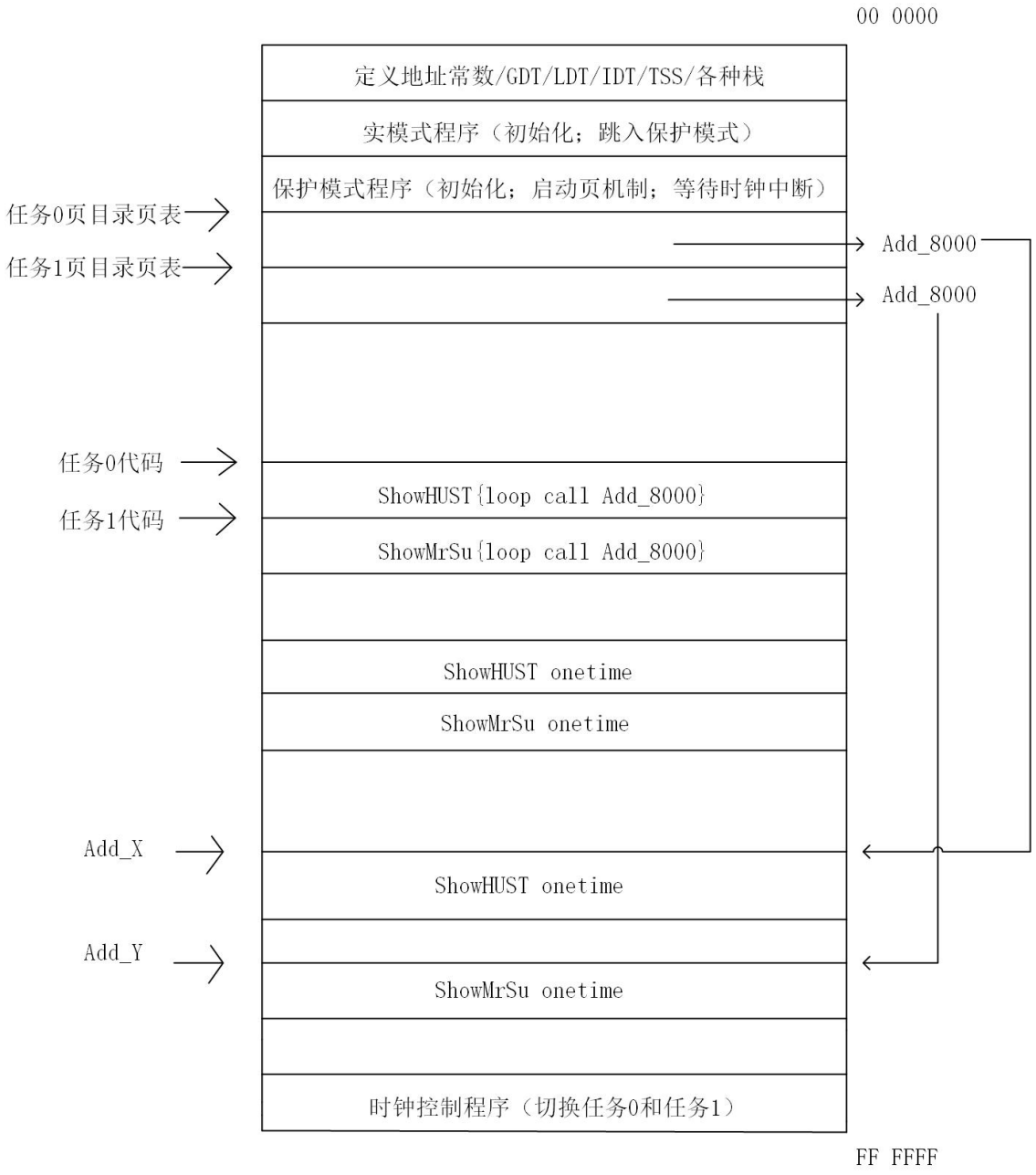

图 4-1 程序内存分布示意图

在实现 task4 多任务调度的任务中，首先需要定义四个任务，并为每个任务分配不同的优先级。在时钟中断服务程序中，采用优先级调度算法来实现多任务的切换。每个任务将被设计为一个死循环，分别循环输出不同的字符串。任务的输出持续时间与其优先级相对应，优先级越高的任务运行机会越多，输出时长也就越长。

实验流程大致如下：

创建全局描述符表 GDT，进行段描述符和选择子的初始化。创建中断描述符表 IDT，将中断向量与其对应的门选择子和偏移进行关联。对任务状态段 TSS 进行初始化操作。创建任务局部描述符表 LDT，对其任务数据段、代码段和堆栈段等描述符和选择子进行初始化，将段基址装入相应的段描述符内。对各个任务的页目录表和页表进行初始化，以便实现任务间的内存隔离和管理。初始化 8259A 可编程中断控制器、8253A 可编程定时器以及 IDT。设置 IRQ0 以配置时钟中断，并配置时钟中断的触发周期。编写时钟中断服务程序和调度算法，在每次时钟中断时选择应该被调度的任务。根据不同特权级的跳转来切换到对应的任务，每个任务实现循环输出特定的字符串的操作。确保每个任务都会在其优先级所允许的范围内执行，并按照预期的方式进行切换和输出。

4.2 实验重点原理

在进入保护模式前的准备过程中，刚开始执行的程序尚未进行其他操作。此时，需要初始化各个描述符的基址。在这个阶段，段寄存器 DS、ES、SS、CS 都被设置为相同的值。初始化各个描述符基址的代码具有重复性，都需要通过实模式下的寻址方式计算出各个段的基址，并将其填入描述符的相应部分。为了简化这个过程，可以利用 NASM 中的宏，例如可以将初始化段描述符放入宏 InitDescBase，该宏定义在 pm.inc 中，下面是该宏的代码：

```
; 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
```

使用宏可以显著减少代码冗余性，使得初始化描述符的基址更加简洁和高效。同理在定义任务、初始化任务描述符、代码段、数据段和堆栈时也可以利用宏定义简化代码，不仅可以减少代码量，也使得代码的结构更加清晰。

为了确保程序运行结果不受程序运行前的显示信息的影响，在输出信息之前，需要首先执行清屏操作。通过调用函数 ClearScreen 可以将屏幕上所有区域设置为黑底黑字，不显示任何信息，从而避免了之前信息的干扰。函数 ClearScreen 位于 lib.inc 内，其代码如下：

```
ClearScreen:        ; 清屏操作
  push  eax
  push  ebx
  push  ecx
  mov       ah, 00000000b     ; 0000: 黑底    0000: 黑字
  mov       al, 0
  mov       ebx, 0
  mov       ecx, 4000
.1:
  mov       [gs:ebx], ax
  add       ebx, 2
  loop  .1
  pop       ecx
  pop       ebx
  pop       eax
  ret
```

4.3 实验关键代码

跨特权级的任务由于需要不同特权级之间的转换，不能简单地使用 jmp 或 call 指令进行跨特权级任务的调用。这种情况可以使用 IRETD 指令来进行特权级的变换，实现从低特权级到高特权级的转换。

IRETD 指令通常用于从中断或异常处理程序返回到用户态代码，并将控制权交给一个新的代码段和堆栈。它会从堆栈中弹出 EIP 寄存器的值，该值指向新代码段中的指令。同时，IRETD 指令还会弹出 CS 寄存器的值，该值是指向新代码段选择器的指针，该选择器包含了新代码段描述符在 GDT 表中的偏移量。该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。最后，IRETD 指令从堆栈中弹出一个新的堆栈选择器和堆栈指针，以便切换到新的堆栈。

所以在使用 IRETD 进行任务调用跳转时，需要依次将任务对应的 SS、ESP、EFLAGS、CS 和 EIP 压入堆栈中，以便正确地进行地址寻址。

以下是使用宏的代码示例：

```
push   SelectorTask%1Stack3    ; 压入新的堆栈选择器
push    TopOfTask%1Stack3   ; 压入新的堆栈指针
pushfd                      ; 压入 EFLAGS 寄存器的值
pop     eax                 ; 弹出并保存 EFLAGS 寄存器的值到 eax
or      eax, 0x200          ;将 EFLAGS 中的 IF 位置 1，开启中断
push    eax                 ; 将修改后的 EFLAGS 重新压入堆栈
push    SelectorTask%1Code  ; 压入新的代码段选择器
push    0                   ; 压入新代码段的偏移地址
```

该代码的作用是将任务对应的堆栈选择器、堆栈指针、修改后的 EFLAGS、新的代码段选择器和新代码段的偏移地址依次压入堆栈中，以准备进行特权级的转换和任务调用跳转。

在 task4 中，任务调度算法是最为关键而且最困难的部分。任务指导书以及

老师的讲解中都没有对这一部分进行详细的介绍，因此我刚开始尝试时陷入了一个误区，我一直在思考优先级和门以及时钟中断该如何关联起来，却没有找到任何的头绪，因此在第二周实验验收时只完成了 task3。在之后我通过查找资料和询问同学，才明白应该将它当作单独的一个算法分开来看，最终成功实现了优先级调度算法，完成了 task4。

任务优先数调度算法的思路如下：

在创建进程时，分别给每个任务不同的优先级数，优先级高的任务优先数更大。

每次调度时，会先判断当前任务的优先级数是否为 0。若不为 0，则继续执行该任务；若为 0，则在其他任务中选择优先级数最高的任务执行。

若所有任务的优先数均为 0，则进入新的一轮调度，赋予每个任务初始的优先级数。

代码框架如下：

```
ClockHandler:
    push    ds
    pushad
    ; 通知 PIC 芯片已经处理完时钟中断
    mov     al, 0x20
    out     0x20, al
    ; 检查前一个任务的 ticks 是否为 0
    ; 如果不为 0，直接跳转到 subTicks
    ; 如果为 0，则进行任务选择和切换
    ; 如果所有任务的 ticks 均为 0，则重新赋值 ticks
    ; 最后，结束时钟中断处理过程
    ; 任务选择和切换
    ; 减少 ticks 操作
    ; 重新赋值 ticks
```

```
    ; 结束时钟中断处理过程
    popad
    pop     ds
    iretd
```

5 开发和运行环境的配置

开发环境：Windows11，汇编器 nasm，vscode

运行环境：Ubuntu 16.04.7，汇编器 nasm，虚拟机 bochs-2.3.5，freedos 软盘映像文件

运行环境的配置：

Bochs 的安装，从官网上下载 2.3.5 安装包，解压后利用 ./configure --enable-debugger --enable-disasm 指令打开调试功能，之后进行安装。

在官网下载 freedos 软盘映像文件，解压后内含 4 个文件，保留 a.img 并更名为 freedos.img 备用，作为软盘 A。用 bximage 工具制作 pmtest.img，作为软盘 B。

配置运行控制文件 bochsrc.txt，

```
megs: 32
# filename of ROM images
romimage: file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/share/bochs/VGABIOS-lgpl-latest
# what disk images will be used
floppya: 1_44=freedos.img, status=inserted
floppyb: 1_44=pm.img, status=inserted
# choose the boot disk.
boot: a
# disable the mouse
mouse: enabled=0
```

更新程序：

使用 mount 将格式化的 pmtest.img 软盘映像挂接到一个目录，将更新的程序复制到挂载点下，在更新完成后卸载软盘镜像，启动 Bochs 并加载更新后的软盘镜像 B 即可运行更新后的程序。

运行程序：

在虚拟机中该文件夹下执行 bochs -f bochsrc 指令打开虚拟机测试运行；输入 c 进入 A 盘，然后输入 B:进入 B 盘，输入程序名称运行

调试程序：

在 Bochs 配置文件中设置调试选项 magic_break: enabled=1，在命令行中输入 bochsdbg 启动 Bochs 调试器，在调试器中，使用 floppy 命令加载软盘 B，使用 b 命令设置断点，使用 c 命令开始执行程序。Bochs 将会执行到设置的断点处停止，程序停止执行后可以使用命令来查看和修改内存、寄存器等内容，以及单步执行指令，在调试完成后使用 q 命令退出 Bochs 调试器。

观察程序的运行效果：

观察 Bochs 的控制台窗口中打印的信息、程序窗口以及图形界面中显示的内容来判断程序的任务切换状态。

6 运行和测试过程

VERY 为第一个任务的输出，由于其优先数最大，运行机会多，可以观察到 VERY 显示时间最长。

图 6-1 任务一运行图

LOVE 为第二个任务的输出，优先数第二大，显示时间第二长。

图 6-2 任务二运行图

HUST 为第三个任务的输出，优先数第三大，显示时间第三长

图 6-3 任务三运行图

MRSU 为最后一个任务的输出，由于其优先数最小，运行机会少，可以观察到 MRSU 显示时间最短。

图 6-4 任务四运行图

综上，可以观察到每个任务都是死循环，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。根据时钟中断服务程序，选择任务，切换任务。

7 实验心得和建议

在本次实验中，我根据实现指导书一步步实现了一个简单的优先数调度的多任务操作系统，在实验中遇到的一些问题及解决方法如下：

(1) 问题：bochs 虚拟机开机后显示无可引导设备

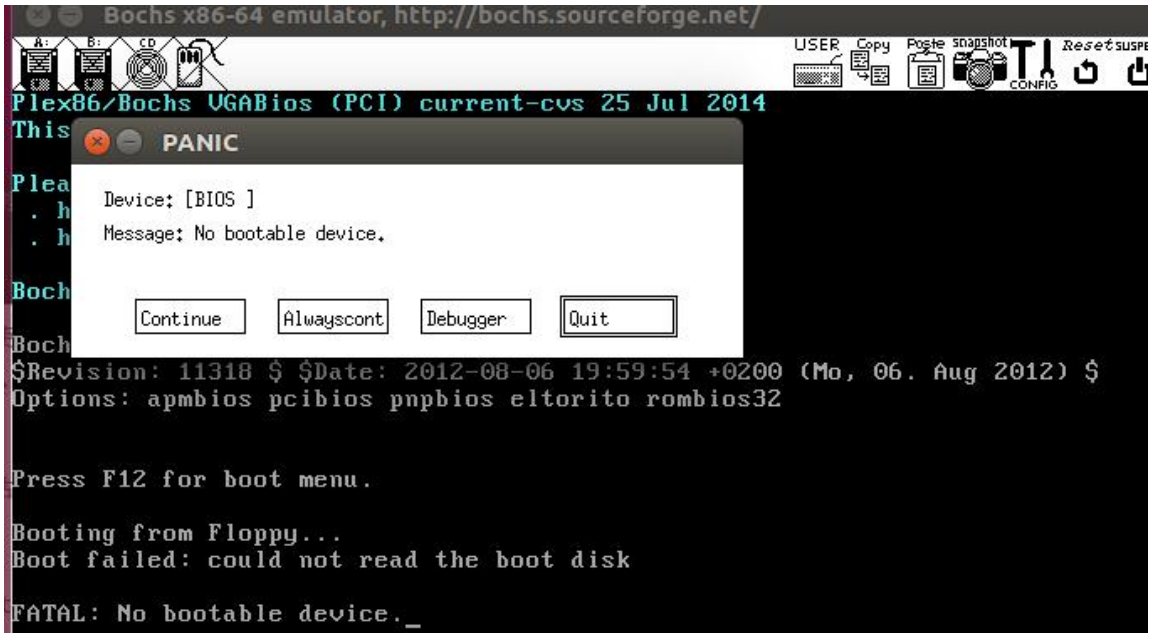

图 7-1 bochs 故障示意图

解决方法：使用 wget http://bochs.sourceforge.net/guestos/freedos-img.tar.gz 安装的文件不完整，应该从官网上下载完整的压缩包解压，然后进行参数的配置开启调试功能。

(2) 问题：进行初始化时，代码存在大量重复，过于冗余不利于编写和调试

解决方法：使用宏可以显著减少代码冗余性，使得初始化更加简洁和高效。在定义段描述符，初始化任务描述符、代码段、数据段和堆栈时可以利用宏定义简化代码，不仅可以减少代码量，也使得代码的结构更加清晰。

(3) 问题：从用户态的一个任务跳转到另一个任务时出现问题

解决方法：在内核态的代码中调用系统服务例程，来进一步处理用户态的请求，检查权限并跳转到另一个用户态代码段，继续执行相应的任务操作。

(4) 问题：Ring3 代码段运行时出现闪退，无法看到运行效果。

解决方法：首先检查退出保护模式时是否正确关闭了分页机制，确保所有相关的段和选择子属性设置正确，VIDEO 段需要设置为 DPL3 属性以允许 Ring3 代码段访问。

(5) 问题：优先级调度算法思路较为复杂，在实现时容易出现一些细节问题。

解决方法：先写出清晰的伪代码框架，再进行填充。

建议：

《Orange'S：一个操作系统的实现》这本书中关于特权级的讲解比较复杂，而且没有简单的实现方法的示例，作为初学者看起来比较迷茫没有思路，建议实验指导书可以增加有关特权级的解释说明，希望老师能够对特权级及其转换做出更易于理解的讲解

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

（4）《Orange'S：一个操作系统的实现》