
華中科技大學

操作系统课程设计报告

课 程： 操作系统课程设计

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 2204 班

学 号： U202212021

姓 名： 戴申宇

2025 年 03 月 28 日

目 录

1	课设目的.....	1
2	课程设计内容.....	4
3	程序设计思路.....	5
	3.1 初始化模块	5
	3.2 任务定义和数据结构模块	7
	3.3 任务切换模块	9
	3.4 中断处理模块	11
	3.5 分页机制模块	13
	3.6 辅助功能模块	15
4	实验程序的难点或核心技术分析.....	16
	4.1 保护模式的切换	16
	4.2 任务的定义和切换	17
	4.3 时钟中断处理	19
5	开发和运行环境的配置	22
	5.1 开发环境配置	22
	5.2 运行环境配置 (bochsrc.txt).....	23
	5.3 更新程序	23
	5.4 运行/调试程序	24
6	运行和测试过程.....	25
7	实验心得和建议	27
8	学习和编程实现参考	29

1 课设目的

1. 深入理解保护模式基本工作原理：

保护模式是 x86 架构下的一种重要运行模式，它提供了内存保护、多任务支持等关键特性，是现代操作系统运行的基础。本次课程设计要求不依赖现成的操作系统，从零开始搭建一个运行在保护模式下的内核雏形。通过这个过程，能够亲身体会到从实模式切换到保护模式的过程，理解保护模式下 CPU 的寄存器结构、控制寄存器（如 CR0）的作用、以及如何通过设置标志位来启用保护模式的各项功能。

2. 透彻理解保护模式地址映射机制：

保护模式下的地址映射是操作系统实现内存管理的关键。本次课程设计要求理解并实现保护模式下的两种主要地址映射方式：段机制和页机制。需要理解线性地址、逻辑地址、物理地址之间的转换关系，掌握如何通过编程配置段描述符、页目录、页表等来实现地址映射，从而为操作系统提供灵活的内存管理能力。

3. 掌握段机制和页机制的具体实现：

- **段机制：**了解段的概念（代码段、数据段、堆栈段等），理解段描述符的各个字段（段基址、段限长、访问权限等）的含义，以及如何利用全局描述符表（GDT）和局部描述符表（LDT）来管理多个段。此外，还需要了解段选择子的结构及其在寻址过程中的作用。
- **页机制：**理解页的概念（通常为 4KB 大小），了解页目录和页表的层级结构，以及页目录项（PDE）和页表项（PTE）中各个字段（物理页框地址、访问权限、存在位等）的含义。通过配置页目录和页表，可以实现线性地址到物理地址的转换，并为不同的内存区域设置不同的访问权限。

4. 深刻理解任务/进程的概念和切换过程：

多任务是现代操作系统的核心特性之一。本次课程设计要求理解任务（或进程）的概念，了解任务的组成要素（代码、数据、堆栈、任务状态等）。掌握任务状态段（TSS）的结构和作用，理解如何通过 TSS 来保存和恢复任务的上下文（寄存器状态）。此外，还需要理解局部描述符表（LDT）在多任务环境中的作用，每个任务可以拥有自己的 LDT，从而实现任务间的隔离。通过手动创

建 TSS 和 LDT，并利用中断机制（时钟中断）来实现任务的切换，深入理解操作系统任务调度的底层原理。

5. 掌握优先数进程调度原理：

本次课程设计要求实现一种基于优先数的进程调度算法。理解优先数调度算法的基本思想：为每个任务分配一个优先数，调度器总是选择就绪队列中优先数最高的任务运行。优先数可以根据任务的类型、重要性等因素来确定。通过实现优先数调度算法，了解操作系统如何根据任务的优先级来合理分配 CPU 时间，提高系统资源的利用率。

6. 编程技能培养：

通过本次课程设计，底层编程能力得到显著提升，体现在以下几个方面：

- **掌握保护模式的初始化：**如何编写汇编代码，完成从实模式到保护模式的切换，配置 GDT、IDT 等关键数据结构，并启用分页机制。
- **掌握段机制的实现：**根据需要定义不同的段（代码段、数据段等），设置段描述符的各个字段，并在 GDT 或 LDT 中注册这些段。
- **掌握页机制的实现：**创建页目录和页表，填写 PDE 和 PTE，将线性地址空间映射到物理内存，并设置合适的访问权限。
- **掌握任务的定义和任务切换：**定义任务的数据结构（TSS、LDT），编写任务的代码，并在时钟中断处理程序中实现任务的切换。
- **掌握保护模式下中断程序设计：**设置中断描述符表（IDT），编写中断处理程序，并在中断处理程序中完成必要的操作（如保存和恢复任务上下文、处理设备请求等）。

7. 掌握保护模式的初始化编程：

从实模式到保护模式的平稳过渡是操作系统启动的关键步骤。本次课程设计要求编写汇编代码，实现这一关键性的切换。这包括设置控制寄存器（如 CR0）、加载全局描述符表寄存器（GDTR）和中断描述符表寄存器（IDTR）、以及处理好地址线 A20 的开启等细节问题。通过这一过程，对保护模式的启动过程有一个清晰的认识。

8. 熟练掌握段机制的实现：

段机制是保护模式下内存管理的基础。本次课程设计要求根据实验需求，定义各种类型的段（如代码段、数据段、堆栈段），并正确设置段描述符的各个字段（包括段基址、段限长、访问权限等）。理解不同类型的段描述符的差

异，并在全局描述符表（GDT）或局部描述符表（LDT）中正确地注册这些段。

9. 精通掌握页机制的实现：

页机制提供了更细粒度的内存管理和保护能力。本次课程设计要求手动创建页目录和页表，并正确填写页目录项（PDE）和页表项（PTE）。这包括理解 PDE 和 PTE 中各个位（如存在位、读写权限位、用户/超级用户位等）的含义，以及如何根据需要设置这些位。通过实现页机制，将线性地址空间映射到物理内存，并为不同的内存区域设置不同的访问权限，从而实现内存保护和虚拟内存。

10. 深入掌握任务的定义和任务切换：

多任务是现代操作系统的核心特性。本次课程设计要求定义任务的数据结构，包括任务状态段（TSS）和局部描述符表（LDT）。理解 TSS 中各个字段的含义，以及如何利用 TSS 来保存和恢复任务的上下文（寄存器状态）。此外，需要编写任务切换的代码，通常在时钟中断处理程序中实现。通过手动实现任务切换，深入理解操作系统任务调度的底层原理。

11. 熟练掌握保护模式下的中断程序设计：

中断是操作系统响应外部事件和实现系统调用的重要机制。本次课程设计要求设置中断描述符表（IDT），并编写相应的中断处理程序。需要理解中断描述符的结构，以及如何在 IDT 中注册中断处理程序。此外，还需要了解中断处理程序的编写规范，包括如何保存和恢复现场、如何处理中断请求、以及如何返回到被中断的程序。

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）。

- 裸机(bochs)中启动保护模式，创建多个不同优先级的任务（每个任务死循环在屏幕相同位置输出不同字符串，（含TSS，LDT，代码，堆栈，页目录/页表等要素））。所有任务在时钟（周期50ms，可调）驱动下进行调度，调度策略采用“优先数进程调度策略”。
- 示例：4个任务（优先级分别为16,10,8,6）各自输出：**VERY** **LOVE** **HUST** **MRSU** 字符串，每个字符串的持续显示时间长度有差异（体现了优先级的差异）【参考：课设效果演示视频1.mp4】。可以观察到VERY显示时间最长，MRSU最短，LOVE，HUST居中。

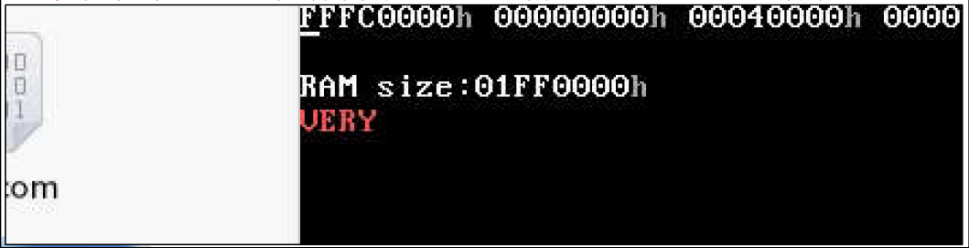
The image shows a screenshot of a Bochs virtual machine window. On the left, there's a sidebar with a floppy disk icon and the text 'com'. The main window displays memory addresses in hexadecimal: 'FFFC0000h 00000000h 00040000h 0000'. Below these, it says 'RAM size: 01FF0000h'. At the bottom, the word 'VERY' is displayed in red text.

图 2-1 课程设计任务

3 程序设计思路

本程序在裸机环境下构建了一个支持多任务切换的、基于 x86 保护模式的简易程序。程序采用汇编语言编写，从逻辑上我将其分为共 6 个模块，总的模块设计和逻辑关系如下 UML 图所示：

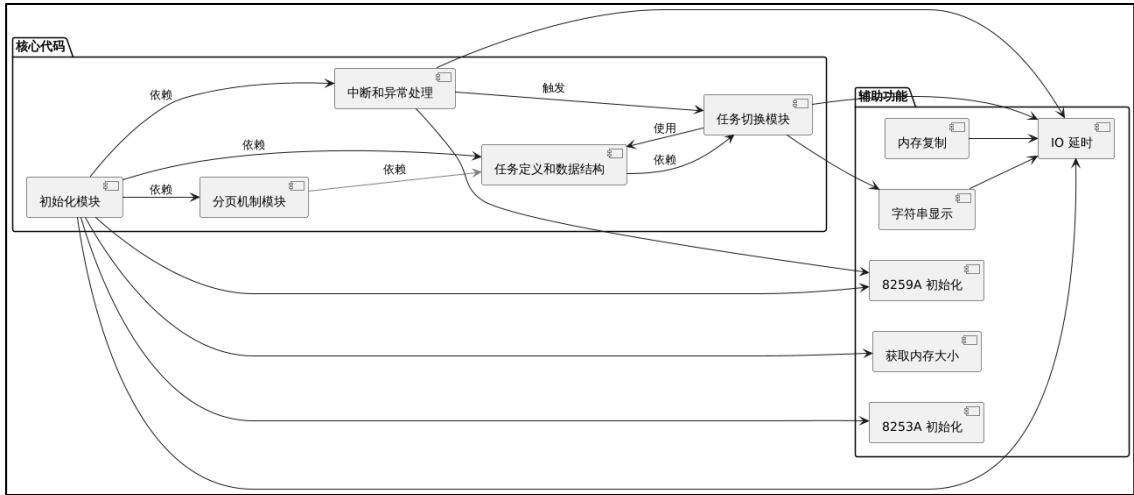


图 3-0 模块设计示意图

3.1 初始化模块

。 概念和原理：

- **实模式与保护模式：** x86 CPU 启动时处于实模式，实模式下内存寻址采用段基址:段内偏移地址的方式，最大寻址空间为 1MB。保护模式提供更强大的内存管理和保护机制，支持更大的寻址空间（32 位下为 4GB）和多任务。
- **全局描述符表（GDT）：** GDT 是保护模式下用于存储段描述符的数组。段描述符定义了段的属性，如基地址、限长、访问权限等。CPU 通过 GDT 来查找段描述符，从而实现对内存段的访问。

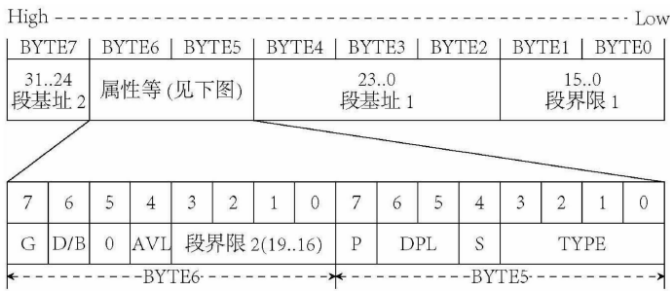


图 3-1 描述符结构

- **中断描述符表 (IDT):** IDT 是保护模式下用于存储中断门描述符的数组。中断门描述符定义了中断处理程序的入口地址等信息。当发生中断或异常时, CPU 通过 IDT 来查找对应的中断处理程序。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
描述符索引													TI	RPL	

图 3-2 选择子结构

- **控制寄存器 (CR0):** CR0 寄存器包含了一些控制 CPU 运行模式的标志位, 如 PG 位 (启用分页)、PE 位 (启用保护模式) 等。
 - **A20 地址线:** 由于历史原因, 早期 x86 CPU 的地址线只有 20 根, 最大寻址空间为 1MB。为了兼容旧的程序, 在实模式下, A20 地址线默认是关闭的。进入保护模式后, 需要手动打开 A20 地址线, 才能访问 1MB 以上的内存。
- 。 设计和实现思路:

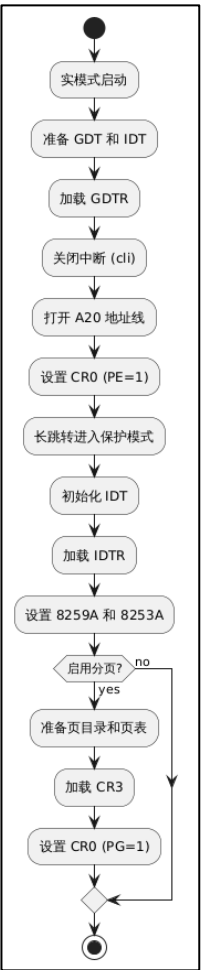


图 3-3 初始化模块流程

i. 从实模式进入保护模式：

1. 保存实模式下的状态（如段寄存器、堆栈指针等）。
2. 准备 GDT：定义 GDT 中的段描述符（包括代码段、数据段、堆栈段等）。
3. 加载 GDTR：将 GDT 的基地址和限长加载到 GDTR 寄存器。
4. 关闭中断：通过 cli 指令关闭中断，防止在切换过程中发生意外。
5. 打开 A20 地址线：通过向 0x92 写入值来打开 A20 地址线。
6. 设置 CR0 寄存器：将 PE 位置为 1，启用保护模式。
7. 使用长跳转指令 jmp dword SelectorCode32:0 进保护模式代码段。

ii. 初始化 IDT：

1. 定义 IDT。
2. 定义中断门描述符。
3. 加载 IDTR：将 IDT 的基地址和限长加载到 IDTR 寄存器。

iii. 设置 8259A 和 8253A：

1. 8259A 是可编程中断控制器，用于管理硬件中断。
2. 8253A 是可编程间隔定时器，用于产生时钟中断。
3. 对这两个芯片进行初始化，设置中断向量、中断屏蔽。

iv. 启用分页机制：

1. 准备页目录和页表，并将页目录的基地址加载到 CR3 寄存器。
2. 将 CR0 寄存器的 PG 位置为 1，启用分页机制。

3.2 任务定义和数据结构模块

○ 概念和原理：

- **任务 (Task)：**在操作系统中，任务是程序执行的实例。每个任务都有自己的代码、数据、堆栈以及其他资源。
- **任务状态段 (TSS)：**TSS 是 x86 架构中用于保存任务上下文（寄存器状态）的数据结构。当发生任务切换时，CPU 会将当前任务的上下文保存到 TSS 中，并从目标任务的 TSS 中恢复上下文。

- **局部描述符表 (LDT):** LDT 类似于 GDT，但它是每个任务私有的。LDT 中存储了任务私有的段描述符，如任务的代码段、数据段、堆栈段等。通过使用 LDT，可以实现任务之间的隔离，防止它们相互干扰。
 - **任务切换:** 任务切换是指操作系统将 CPU 的控制权从一个任务转移到另一个任务的过程。任务切换通常由时钟中断触发。
- **设计和实现思路:**
- i. **定义 TSS 结构:**
1. 根据 x86 架构的要求，定义 TSS 的各个字段，包括通用寄存器 (EAX、EBX、ECX、EDX 等)、段寄存器 (CS、DS、ES、SS 等)、EFLAGS 寄存器、EIP 寄存器、以及指向 LDT 的指针等。
 2. 为每个任务创建一个 TSS 实例。

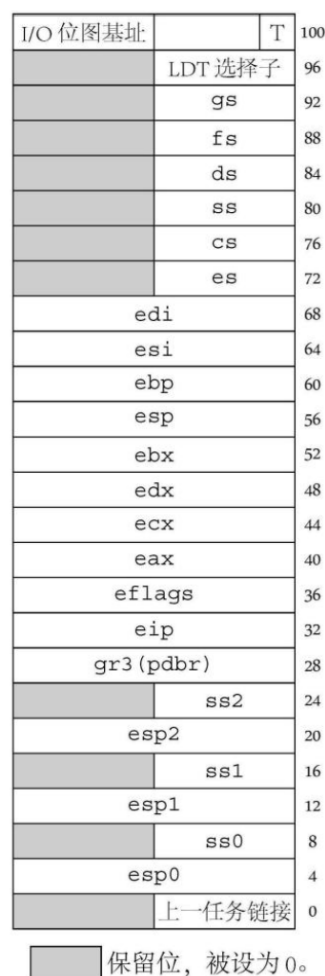


图 3-4 TSS 结构示意图

ii. 定义 **LDT** 结构:

1. 为每个任务定义一个 **LDT**。
2. 在 **LDT** 中定义任务私有的段描述符，如任务的代码段、数据段、堆栈段（用户态和内核态）等。
3. 设置段描述符的各个字段，包括段基址、段限长、访问权限等。确保不同任务的段描述符不会相互冲突。

iii. 任务数据结构:

除了 **TSS** 和 **LDT**，还可以定义其他与任务相关的数据结构，如任务的名称、优先级、状态（就绪、运行、阻塞等）等。

iv. 在 **GDT** 中定义每个任务的 **LDT** 和 **TSS** 描述符

在全局描述符表中，为每个任务的 **LDT** 和 **TSS** 创建相应的描述符。

v. 任务代码和数据:

1. 为每个任务编写代码。任务代码通常是一个循环，执行一些操作（如打印字符）。
2. 为每个任务分配数据空间，用于存储任务的变量。

3.3 任务切换模块

。 概念和原理:

- **任务切换**: 任务切换是操作系统将 **CPU** 控制权从一个任务转移到另一个任务的过程。这是实现多任务的关键机制。
- **时钟中断**: 时钟中断是由硬件定时器（如 8253/8254 芯片）周期性产生的。操作系统通常利用时钟中断来触发任务切换，保证每个任务都能获得 **CPU** 时间。
- **中断处理程序**: 中断处理程序是响应特定中断的例程。时钟中断处理程序负责处理时钟中断，并在必要时执行任务切换。
- **任务调度**: 任务调度是指操作系统根据一定的策略（如优先级调度、轮转调度等）选择下一个要运行的任务的过程。
- **TSS 和 LDT 的作用**: 在任务切换过程中，**CPU** 使用 **TSS** 来保存和恢复任务的上下文。**LDT** 则用于隔离不同任务的地址空间。

。 设计和实现思路:

i. 时钟中断处理程序：

1. 在 IDT 中注册时钟中断处理程序。
2. 时钟中断处理程序的主要职责：
 - a) 保存当前任务的上下文（寄存器状态）到其 TSS 中。
 - b) 调用任务调度器选择下一个要运行的任务。
 - c) 从目标任务的 TSS 中恢复上下文。
 - d) 切换 LDT（每个任务有独立的 LDT）。
 - e) 使用 iret 指令返回到目标任务的代码继续执行。

ii. 任务调度器：

1. 实现一个任务调度器，负责根据策略选择下一个要运行的任务。
2. 本实验中采用优先级调度策略，选择优先级最高的任务。
3. 调度器需要维护一个就绪队列，记录所有就绪的任务。
4. 在每次时钟中断时，调度器会检查当前任务的剩余时间片。如果时间片用完，或者有更高优先级的任务就绪，则进行任务切换。

iii. 任务切换的具体步骤：

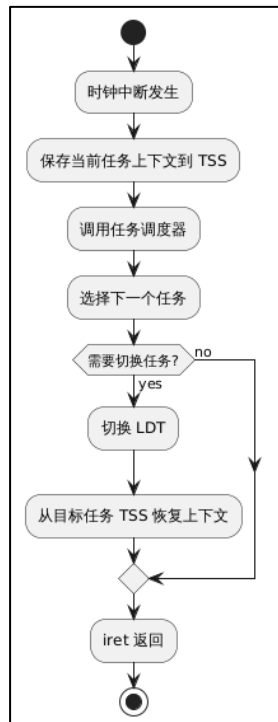


图 3-5 任务切换模块流程

-
1. **保存现场：**将当前任务的通用寄存器、段寄存器、EFLAGS 寄存器、EIP 寄存器等保存到其 TSS 中。
 2. **切换 LDT：**通过 lldt 指令加载目标任务的 LDT 选择子，切换到目标任务的地址空间。
 3. **加载 TSS：**CPU 自动从目标任务的 TSS 中加载上下文
 4. **恢复现场：**从目标任务的 TSS 中恢复通用寄存器、段寄存器、EFLAGS 寄存器、EIP 寄存器等。
 5. **返回执行：**使用 iret 指令从中断处理程序返回，CPU 将从目标任务的 EIP 寄存器指向的地址继续执行。

3.4 中断处理模块

。 概念和原理：

- **中断：**中断是外部硬件设备向 CPU 发出的信号，请求 CPU 暂停当前执行的任务，转而去处理中断事件。
- **中断向量：**中断向量是一个整数，用于标识不同类型的中断或异常。每个中断或异常都有一个唯一的中断向量。
- **中断描述符表 (IDT)：**IDT 是一个数组，用于存储中断门描述符或陷阱门描述符。每个中断或异常在 IDT 中都有一个对应的描述符，描述符中包含了中断处理程序的入口地址等信息。
- **中断门和陷阱门：**中断门和陷阱门是两种不同类型的门描述符。中断门用于处理硬件中断，陷阱门用于处理异常和软中断。
- **中断处理程序：**中断处理程序是响应特定中断或异常的例程。当发生中断或异常时，CPU 会根据中断向量在 IDT 中查找对应的描述符，跳转到描述符中指定的中断处理程序入口地址执行。

。 设计和实现思路：

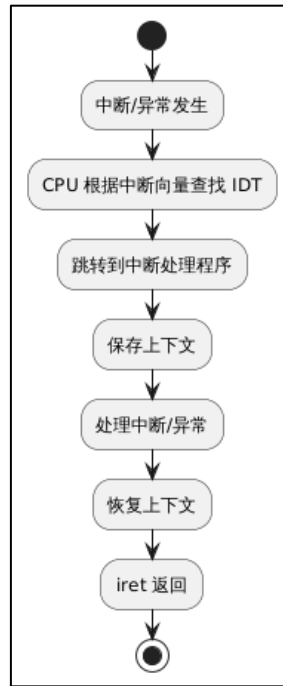


图 3-6 中断处理模块流程

i. 初始化 IDT:

1. 在内存中创建一个 IDT 数组。
2. 为每个需要处理的中断或异常在 IDT 中创建一个对应的描述符（中断门或陷阱门）。
3. 设置描述符的各个字段，包括中断处理程序的入口地址、段选择子、DPL（描述符特权级）等。
4. 将 IDT 的基地址和限长加载到 IDTR 寄存器。

ii. 编写中断处理程序:

1. 为每个需要处理的中断或异常编写一个中断处理程序。
2. 中断处理程序的主要职责是：
 - a) 保存被中断任务的上下文（如果需要）。
 - b) 处理中断或异常事件。
 - c) 如果需要，可以进行任务切换。
 - d) 恢复被中断任务的上下文（如果需要）。
 - e) 使用 `iret` 指令返回到被中断的任务继续执行。

3.5 分页机制模块

- 概念和原理：

- **分页机制：**分页机制是 x86 保护模式下的一种内存管理机制，它将线性地址空间划分为固定大小的页（通常为 4KB），并将物理内存也划分为同样大小的页框（Page Frame）。通过页目录和页表，可以将线性地址转换为物理地址。
- **线性地址：**程序使用的地址，一个 32 位的无符号整数。
- **物理地址：**物理地址是 CPU 实际访问的内存地址。
- **页目录：**页目录是一个特殊的页（4KB），它包含了 1024 个页目录项（PDE）。每个 PDE 指向一个页表。
- **页表：**页表也是一个特殊的页（4KB），它包含了 1024 个页表项（PTE）。每个 PTE 指向一个物理页框。
- **CR3 寄存器：**CR3 寄存器存储了当前页目录的物理基地址。
- **地址转换：**线性地址三部分：页目录索引（高 10 位）、页表索引（中间 10 位）、页内偏移（低 12 位）。首先 CR3 寄存器找到页目录，然后线性地址的高 10 位找到对应的 PDE，从 PDE 中取出页表的物理基地址；再根据线性地址的中间 10 位找到对应的 PTE，从 PTE 中取出物理页框的物理基地址；将物理页框的基地址与线性地址的低 12 位偏移相加，得到最终的物理地址。

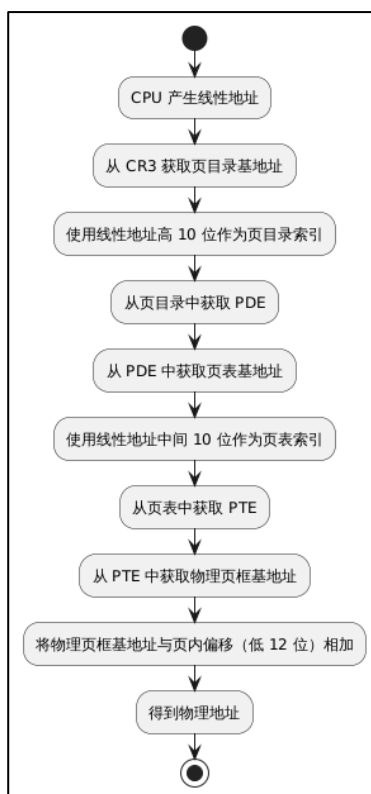


图 3-7 地址转换流程

。 设计和实现思路：

i. 创建页目录和页表：

1. 在内存中分配空间，用于存储页目录和页表。
2. 根据需要，创建多个页表。每个任务需要一个页表来映射其自身的地址空间。

ii. 填写 PDE 和 PTE：

1. 根据线性地址到物理地址的映射关系，填写 PDE 和 PTE。
2. 设置 PDE 和 PTE 中的访问权限位（读/写、用户/超级用户）、存在位等。
3. 对于不存在的页，将 PDE 或 PTE 的存在位设置为 0。

iii. 加载 CR3 寄存器：

将页目录的物理基地址加载到 CR3 寄存器。

iv. 启用分页机制：

将 CR0 寄存器的 PG 位设置为 1，启用分页机制。

v. **线性地址映射设计：**

1. 实现线性地址到物理地址的一一映射
2. 实现不同任务的地址映射
3. 不同任务可以有相同线性地址，但映射到不同的物理地址

3.6 辅助功能模块

○ **代码组织：**

- **模块化：**将代码按照功能划分为不同的模块，辅助模块放在单独的文件中（`pm.inc` 和 `lib.inc`）。
- **宏定义：**使用宏定义来简化代码，提高可读性和可维护性（`Descriptor`、`Gate`、`DefineTask` 等宏）。

○ **字符串显示函数：**

- 由于是在裸机环境下，没有现成的库函数可用，需要自己实现字符串显示函数。
- 该函数将字符串写入显存的特定位置，从而在屏幕上显示出来。
- 可以支持换行、设置字符颜色等功能。

○ **内存复制函数：**

- 用于在内存之间复制数据。
- 在初始化页表、复制任务代码和数据时会用到。

○ **8253A 初始化函数：**

设置 8253A 芯片的工作模式、计数初值等，以产生时钟中断。

○ **8259A 初始化函数：**

设置 8259A 芯片的中断屏蔽字、中断向量，以正确处理硬件中断。

○ **IO 延时函数：**

- 在对 8259A, 8253A 进行操作时，需要短暂的延时。

4 实验程序的难点或核心技术分析

本次实验的核心在于构建一个基于 x86 保护模式的多任务操作系统内核。在实现过程中，涉及到许多底层的概念和技术，其中在代码实现上，以下是我印象深刻的三个关键点，我将逐个解释并总结最终我的解决流程。

4.1 保护模式的切换

- **原理：** CPU 启动时处于实模式，需要手动切换到保护模式。保护模式提供了内存保护、多任务等特性。切换到保护模式需要设置一系列的寄存器和数据结构，包括 GDT、IDT、CR0 等。

- **流程：**

1. 准备 GDT（全局描述符表）。
2. 加载 GDTR（全局描述符表寄存器）。
3. 关闭中断（cli）。
4. 打开 A20 地址线。
5. 设置 CR0 寄存器（PE 位 = 1）。
6. 长跳转进入保护模式代码段。
7. 初始化 IDT（中断描述符表）。
8. 加载 IDTR（中断描述符表寄存器）。

- **关键代码**（节选自 Multitask.asm）

```
; ... (省略部分代码)
; 准备加载 GDTR
mov eax, ds
shl eax, 4
add eax, LABEL_GDT ; eax <- gdt base address
mov dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt base address
; ... (省略部分代码)
; 加载 GDTR
lgdt [GdtPtr]
; 关闭中断
cli
```

```
; ... (省略部分代码)
; 打开 A20 地址线
in al, 92h
or al, 00000010b
out 92h, al
; 准备切换到保护模式
mov eax, cr0
or eax, 1
mov cr0, eax
; 进入保护模式
jmp dword SelectorCode32:0
; ... (省略部分代码)
;初始化 8259 和 8253
call Init8253A
call Init8259A
; ... (省略部分代码)
```

- **难点：** 正确理解保护模式的切换过程，以及各个寄存器和数据结构的作用。容易混淆各个步骤的顺序，或者遗漏关键步骤。
- **解决方法：** 仔细阅读手册中关于保护模式的章节，理解每个步骤的含义。最重要的是参考已有的代码示例，读懂每一行的功能，并进行调试，观察寄存器的变化，确保每一步都正确执行。

4.2 任务的定义和切换

- **原理：** 任务是操作系统中程序执行的实例。每个任务都有自己的代码、数据、堆栈以及 TSS（任务状态段）和 LDT（局部描述符表）。任务切换是指将 CPU 控制权从一个任务转移到另一个任务。
- **流程：**
 1. 定义 TSS 和 LDT 结构。
 2. 为每个任务创建 TSS 和 LDT 实例。
 3. 在 GDT 中注册 TSS 和 LDT 描述符。
 4. 编写任务切换代码（通常在时钟中断处理程序中）。
 5. 在任务切换时，保存当前任务的上下文到 TSS，加载目标任务的 TSS 和 LDT，恢复目标任务的上下文。

-
- 关键代码 (节选自 pm.inc 和 Multitask.asm):

; pm.inc 中定义宏

```
%macro DefineTask 4
    DefineLDT %1
    DefineTaskCode %1, %3, %4
    DefineTaskData %1, %2
    DefineTaskStack0 %1
    DefineTaskStack3 %1
    DefineTaskTSS %1
%endmacro
```

; ... (省略部分代码)

;Multitask.asm

;LDT 和任务的定义

```
DefineTask 0, "VERY", 15, 0Bh
```

```
DefineTask 1, "LOVE", 15, 0Ch
```

```
DefineTask 2, "HUST", 15, 0Dh
```

```
DefineTask 3, "MRSU", 15, 0Eh
```

; ... (省略部分代码)

;时钟中断处理程序中的任务切换代码:

_ClockHandler:

```
ClockHandler equ _ClockHandler - $$
```

; ... (省略部分代码)

; 切换 LDT

;SwitchTask 是一个宏

```
cmp    edx, 0
je     .switchToTask0
cmp    edx, 1
je     .switchToTask1
cmp    edx, 2
je     .switchToTask2
cmp    edx, 3
je     .switchToTask3
```

;pm.inc

;SwitchTask 宏

```
%macro SwitchTask 1
```

```
    mov ax, SelectorLDT%1
```

```
    lldt ax
    mov eax, PageDirBase%1
    mov cr3, eax
    mov eax, SelectorTask%1Data
    mov ds, eax
    push SelectorTask%1Stack3
    push TopOfTask%1Stack3
    pushfd
    pop eax
    or eax, 0x200
    push eax
    push SelectorTask%1Code
    push 0
    iretd
%endmacro
```

- **难点：**理解 TSS 和 LDT 的作用，以及它们在任务切换过程中的关系。正确编写任务切换代码，确保上下文的正确保存和恢复。
- **解决方法：**仔细阅读手册中关于任务管理和任务切换的章节。参考已有的代码示例，并进行调试，观察 TSS 和 LDT 中的值，确保任务切换正确执行。通过更多自定义宏，如 DefineTask 和 SwitchTask 的设计，也大大降低了主程序代码的冗余程度，程序的观感和可解释性也大大提高。

4.3 时钟中断处理

- **原理：**中断是操作系统响应外部事件和处理错误的重要机制。中断通常由硬件设备产生，异常通常由 CPU 在执行指令过程中产生。
- **流程：**
 1. 初始化 IDT（中断描述符表）。
 2. 为每个中断和异常编写处理程序。
 3. 在 IDT 中注册中断和异常处理程序。
 4. 当发生中断或异常时，CPU 根据中断向量在 IDT 中查找对应的处理程序，并跳转到处理程序执行。
 5. 处理程序保存上下文、处理事件、恢复上下文，然后返回。
- **关键代码**（节选自 Multitask.asm):

```

[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT:
; Gate Selector, Offset, DCount, Attribute
%rep 32
Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep
.020h: Gate SelectorCode32, ClockHandler, 0, DA_386IGate
%rep 95
Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep
.080h: Gate SelectorCode32, UserIntHandler, 0, DA_386IGate
IdtLen equ $ - LABEL_IDT
IdtPtr dw IdtLen - 1
dd 0 ; IDT Base Address
; ... (省略部分代码)
;中断处理程序示例（时钟中断）
_ClockHandler:
ClockHandler equ _ClockHandler - $$
push ds
pushad
; ... (省略部分代码)
;处理时钟中断
    mov     edx, dword [RunningTask]
    mov     ecx, dword [LeftTicks+edx*4]
; ... (省略部分代码)
; 发送 EOI
mov al, 0x20
out 0x20, al
; ... (省略部分代码)

popad
pop  ds
iretd

```

○ **难点和解决方法：**

- **难点 1：** 正确理解中断和异常的类型、来源以及处理流程。区分中断和异常，了解不同类型的中断（如时钟中断、键盘中断）和异常（如除零错误、缺页异常）的处理方式。
- **解决方法 1：**
 1. 详细阅读手册中关于中断和异常处理的章节，特别是“Interrupt and Exception Handling”部分。重点关注中断向量、中断描述符表（IDT）、中断门、陷阱门、中断处理程序的结构等概念。
 2. 查阅资料，了解常见的中断和异常类型及其对应的中断向量。例如，时钟中断通常对应中断向量 0x20，键盘中断通常对应中断向量 0x21。
 3. 理解中断处理程序的编写规范，包括如何保存和恢复现场、如何处理中断请求、如何向中断控制器发送 EOI（End of Interrupt）信号、如何使用 `iret` 指令返回等。
- **难点 2：** 编写中断处理程序时，容易遗漏保存或恢复某些寄存器，导致程序运行错误。特别是对于嵌套中断，需要特别注意上下文的保存和恢复。
- **解决方法 2：**
 1. 编写中断处理程序时，严格按照规范保存所有需要使用的寄存器。通常使用 `pushad` 和 `popad` 指令来保存和恢复通用寄存器。
 2. 段寄存器需要手动使用 `push` 和 `pop` 指令进行保存和恢复。
 3. 中断处理程序中调用了其他函数，需要注意堆栈的使用，避免堆栈破坏。
 4. 使用 Bochs 的内置调试器来跟踪中断处理程序的执行过程，观察寄存器和内存的变化，帮助定位问题。

5 开发和运行环境的配置

总体而言，本实验的开发和运行环境主要基于以下组件：

- 操作系统: Ubuntu 20.10
- 汇编器: NASM
- 虚拟机: Bochs
- 软盘映像: freedos.img (系统盘) 和 pmtest.img (用户程序盘)

5.1 开发环境配置

1. 安装 Ubuntu:

在 VMWare 上安装 Ubuntu 20.10 操作系统。

2. 安装 NASM:

打开终端，使用以下命令安装 NASM 汇编器：`sudo apt-get install nasm`

安装完成后，可以编译运行示例程序 `pmtest1` 验证 NASM 是否安装成功。

3. 安装 Bochs:

使用以下命令安装 Bochs 虚拟机及其相关工具：

```
sudo apt-get install bochs vgabios bochs-x bximage
```

使用 'whereis' 命令确认其安装路径，后面配置 `bochsrc.txt` 文件会用到：

```
whereis bochs
/usr/bin/bochs /usr/lib/bochs /usr/share/bochs /usr/share/man/man1/bochs.1.gz
```

4. 获取 freedos.img:

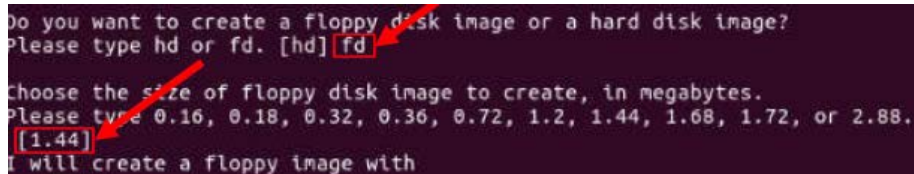
从 Bochs 官网下载 `freedos-img.tar.gz` 文件：

```
wget https://bochs.sourceforge.io/guestos/freedos-img.tar.gz
tar -zxvf freedos-img.tar.gz
```

解压后，将 `a.img` 文件重命名为 `freedos.img`，作为 Bochs 的系统启动盘。

5. 创建 pmtest.img:

使用 `bximage` 工具创建一个 1.44M 的空白软盘映像文件，命名为 `pmtest.img`，用于存放用户程序：



```
Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd

Choose the size of floppy disk image to create, in megabytes.
Please type 0.16, 0.18, 0.32, 0.36, 0.72, 1.2, 1.44, 1.68, 1.72, or 2.88.
[1.44]

I will create a floppy image with
```

图 5-1 创建用户程序盘

按照提示进行创建，选择 fd (软盘)，1.44M，并输入文件名 pmtest.img。

6. 代码文件:

将所有的代码文件(pm.inc, lib.inc, Mutitasking.asm)放到同级目录下。

5.2 运行环境配置 (bochsrc.txt)

Bochs 的配置文件 bochsrc.txt 用于指定虚拟机的各项参数。你需要根据自己的实际情况修改以下配置项：

- **megs:** 虚拟机内存大小，根据需要设置，例如 megs: 32。
- **romimage:** BIOS 映像文件路径，根据 Bochs 的安装路径设置，例如：
romimage: file=/usr/share/bochs/BIOS-bochs-latest。
- **vgaromimage:** VGA BIOS 映像文件路径，例如：vgaromimage:
file=/usr/share/vgabios/vgabios.bin。
- **floppya:** 系统盘 freedos.img 的路径。
- **floppyb:** 用户程序盘 pmtest.img 的路径。
- **boot:** 启动设备，设置为 floppy 从软盘启动。
- **ata0-master:** 如果需要硬盘，配置硬盘映像文件路径。
- **log:** Bochs 日志文件路径。

使用的 bochsrc.txt :

```
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
```

5.3 更新程序

1. 编译:

在 Ubuntu 终端中，进入包含汇编代码的目录，使用 NASM 编译程序：

```
nasm Mutitasking.asm -o Mutitasking.bin
```

其中 Mutitasking 为我的主程序文件。

2. 复制到软盘映像：

首先要将 pmtest.img 挂载到 Ubuntu 的某个目录下，可以使用 mount 命令。

```
mount -o loop ./pmtest.img /mnt/floppyB
```

然后使用以下命令将编译生成的 .com 文件复制到 pmtest.img 映像文件中：

```
sudo cp -r ./Mutitasking.bin /mnt/floppyB
```

5.4 运行/调试程序

1. 运行：

在终端中，使用以下命令启动 Bochs：

```
bochs -f bochsrc.txt
```

Bochs 会加载 freedos.img 启动，然后自动从 pmtest.img 加载程序。

2. 调试：

Bochs 提供了强大的调试功能。在 Bochs 启动后，可以在 Bochs 窗口的命令行中使用以下常用调试命令：

- c (continue): 继续执行。
- s (step): 单步执行。
- b (break): 设置断点，例如 b 0x7c00 在地址 0x7c00 处设置断点。
- x (examine): 查看内存，例如 x /10b 0x7c00 查看从 0x7c00 开始的 10 个字节。
- r (registers): 查看寄存器。
- info registers: 查看所有寄存器内容
- ctrl+c: 中断运行

6 运行和测试过程

一套完整的运行测试流程如下：

首先，在 Bochs 虚拟机中运行起 .com 文件 MT，程序正式开始运行：

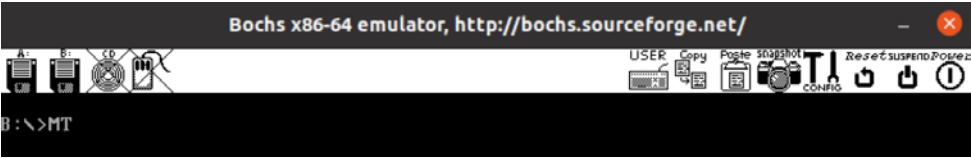


图 6-1 运行程序

随着程序运行，可以逐步观察到效果。

最开始，程序在切换到保护模式后，获取 RAM 大小前，在屏幕顶部设置了一个字符串的回显提示，这里设置是本人的姓名学号：

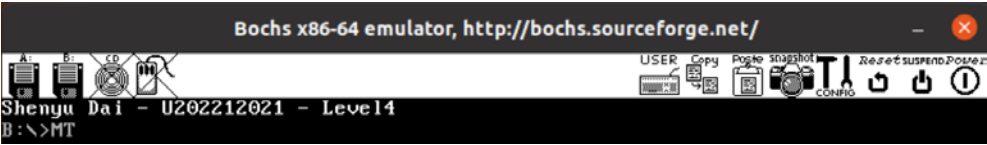


图 6-2 切换到保护模式

然后，获取 RAM 大小的部分程序执行，可以看到运行的结果：

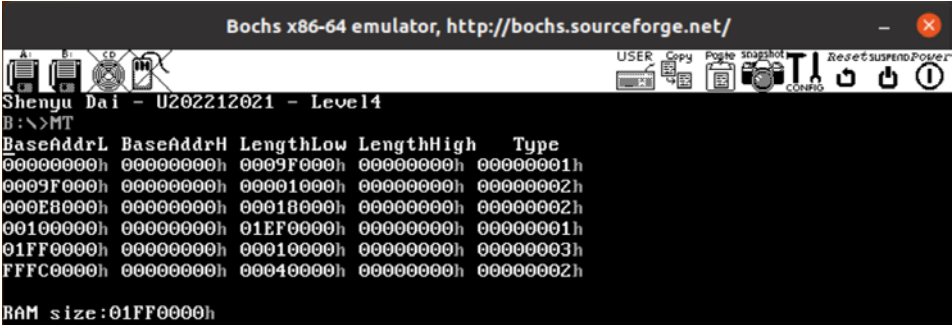


图 6-3 获取 RAM 大小

到此为止，程序准备功能部分的测试全部完成，接下来是主体（四个任务调度）的执行情况显示。

由于 VERY, LOVE, HUST, MRSU 四个字符串的优先数递减（分别为 16, 10, 8, 6），观察到四个字符串在屏幕上的保持时间也是递减的。最后，由于任务调度是一个无限死循环，这四个字符串以下文所述的规律和顺序，循环往复，不断显示出来。具体而言：

① VERY 的持续时间最长——

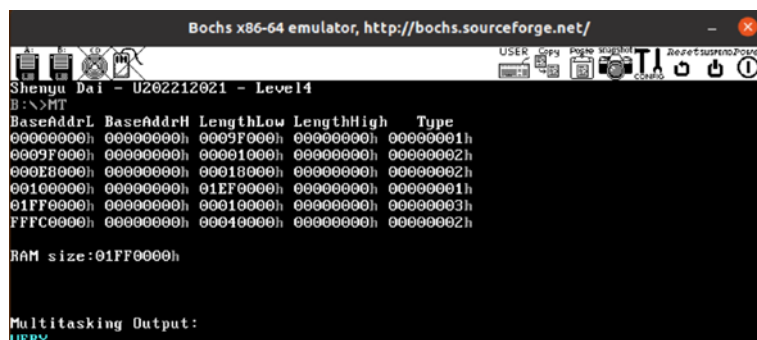


图 6-4 Task1 (优先数 16)

② LOVE 的显示时间其次——

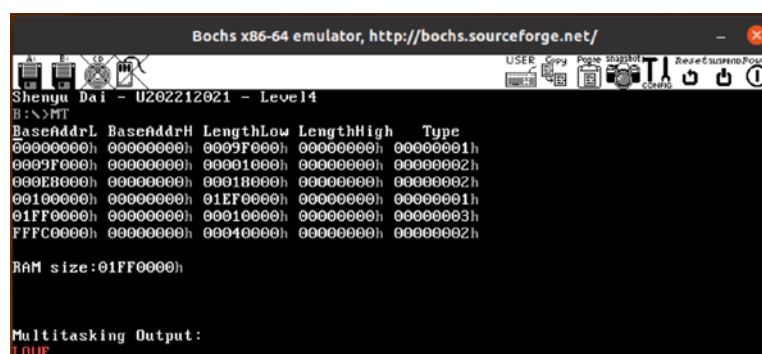


图 6-5 Task2 (优先数 10)

③ HUST 再其次——

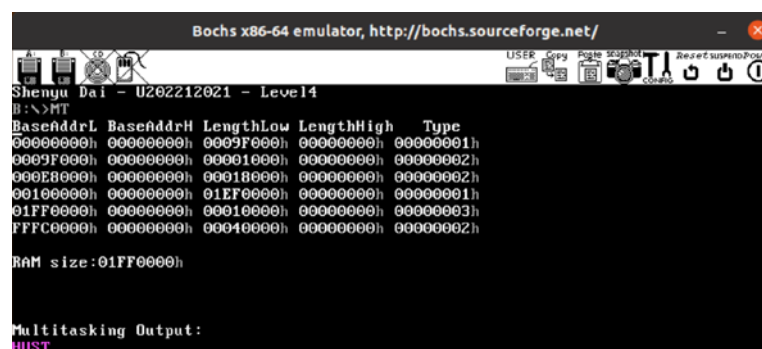


图 6-6 Task3 (优先数 8)

④ 而 MRSU 几乎只是一闪而过——

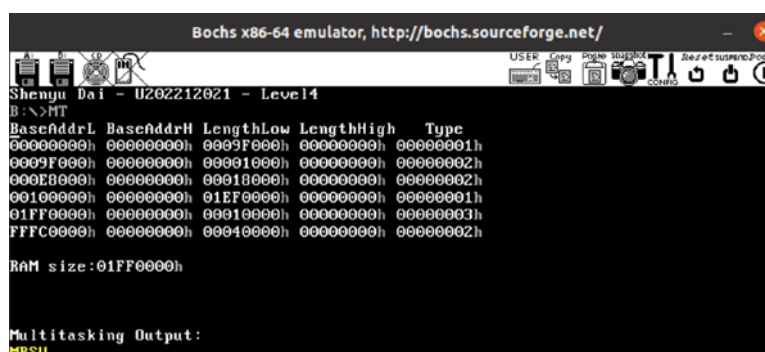


图 6-7 Task3 (优先数 6)

7 实验心得和建议

在本次实验的开发过程中，我遇到过各种各样的问题，通过不断调试和排查，最终解决了这些问题。以下是我遇到的一些问题及其调试过程的记录：

(1) 在实模式下获取内存大小信息时，程序卡死或获取到的信息不正确。

现象：程序在调用 INT 15h 中断获取内存信息时，卡死/返回错误的内存大小。

解决方法：仔细阅读 INT 15h 中断的文档，发现 E820h 子功能需要多次调用才能获取完整的内存映射信息。循环条件 `cmp ebx, 0` 不一定能涵盖所有情况。参考其他资料，了解到需要检查 `int 15h` 返回的进位标志 CF，如果 CF=1，表示获取失败。修改循环条件，根据 `int 15h` 返回的 `ebx` 值判断是否继续循环。只有当 `ebx` 为 0 时，才表示获取完毕。

(2) 在 GDT 中定义段描述符时，段限长计算错误，导致程序访问越界。

现象：程序在访问某个段时，触发一般保护性异常（#GP）。

解决方法：使用 Bochs 调试器查看发生异常时的指令地址和段寄存器的值。发现段限长设置过小，导致程序访问的地址超出了段的范围。

检查段描述符的定义，段限长的单位可以是字节或页（4KB）。如果段描述符中的 G 位（粒度位）为 1，则段限长的单位是页。重新计算段限长。对于代码段，段限长应该是代码段的大小减 1。对于数据段，段限长应该是数据段的大小减 1。如果使用了分页机制，并且段限长设置为 0FFFFFFh，G 位设置为 1，则表示段限长为 4GB。

(3) 在定义 TSS 和 LDT 时，选择子计算错误，导致任务切换失败。

现象：任务切换后，程序立即崩溃或跳转到错误的位置。

解决方法：使用 Bochs 调试器查看任务切换前后的 CS、DS、SS 等段寄存器的值。发现段选择子的值不正确，指向了错误的段描述符。

仔细检查 TSS 和 LDT 的定义，特别是段选择子字段。段选择子的格式为：索引+TI+RPL。确保段选择子的索引值正确指向了 GDT 或 LDT 中对应的段描述符。确保 TI 位正确指示了使用 GDT 还是 LDT（TI=0 表示 GDT，TI=1 表示 LDT）。确保 RPL 设置正确。对于内核代码和数据，RPL 通常设置为 0。对于用户态代码和数据，RPL 通常设置为 3。

(4) 在时钟中断处理程序没有正确恢复寄存器，任务切换后程序状态混乱。

现象：任务切换后，程序行为异常，出现死循环、崩溃。

解决方法：使用 Bochs 调试器单步执行时钟中断处理程序，观察任务切换前后的寄存器状态。发现在中断处理程序中，没有正确保存所有需要使用的寄存器。修改中断处理程序，使用 `pushad` 和 `popad` 指令来保存和恢复所有通用寄存器。对于段寄存器，需要手动使用 `push` 和 `pop` 指令进行保存和恢复。确保在中断处理程序的最后，使用 `iret` 指令返回。

(5) 在实现优先级调度时，没有正确更新任务的剩余时间片，导致高优先级任务一直占用 CPU。

现象：高优先级任务一直运行，低优先级任务没有机会执行。

解决方法：使用 Bochs 调试器观察每个 Task 的 `LeftTicks`。发现在时钟中断处理程序中，没有正确递减当前任务的剩余时间片。修改时钟中断处理程序，在每次时钟中断时，将当前任务的剩余时间片减 1。

当某个任务的剩余时间片减少到 0 时，触发任务切换。为每个任务维护一个时间片计数器，在每次时钟中断时递减该计数器。当计数器减到 0 时，重新加载时间片，并触发任务调度。

(6) 程序中存在硬编码的地址，导致在不同环境下运行时出错。

现象：在不同环境下需要修改多个参数。

解决方法：尽量避免硬编码。使用变量, label 之间的相对地址

(7) Bochs 虚拟机无法正常开机。

现象：报错信息+黑屏

解决办法：Ubuntu22 版本过高不兼容，降级到 20.10 后不再有问题。

8 学习和编程实现参考

- [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] Intel® 64 and IA-32 Architectures Software Developer's Manual:
<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [5] OSDev Wiki: https://wiki.osdev.org/Main_Page
- [6] Bochs 官方网站: <https://bochs.sourceforge.io/>
- [7] 《Orange's: 一个操作系统的实现》
- [8] 南京大学操作系统课程实验（蒋炎岩）: <https://jyywiki.cn/OS/2023/>