|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 操作系统设计与实践 | | | 成 绩 |  | 教师签名 |  |
| 实验名称 | 简单的进程 | | | 实验序号 | 7 | 实验日期 |  |
| 姓 名 | 张子航 | 学 号 | 2021302181026 | | | | 组长 |
| 姓 名 | 辜汝曦 | 学 号 | 2021302141194 | | | | 组员 |
| 姓 名 | 杨馨悦 | 学 号 | 2021302181212 | | | | 组员 |
| 姓 名 | 赵敏 | 学 号 | 2021302181215 | | | | 组员 |

**《操作系统设计与实践》实验报告**

# 实验目的及实验内容

（本次实验所涉及并要求掌握的知识；实验内容等）

## 实验目的

掌握进程的实现机理与进程管理

## 参考资料

《Orange’S 一个操作系统的实现》第六章6.1、6.2、6.3

## 实验要求

* 掌握进程相关数据结构的定义方法：

进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相关的TSS，以及数据结构的关系

* 掌握构造进程的关键技术：

初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动

* 进程的现场保护与切换，弄清楚需要哪些关键数据结构与步骤：

时钟中断与进程调度关系，现场保护与恢复机理，从ring0-->ring1的上下文切换方法，中断重入机理

# 实验环境及实验步骤

（列出本次实验所使用的软件、工具；简要概括实验步骤）

## 实验环境

虚拟机工具：VMWare Workstation 16

虚拟机版本：Ubuntu 14.04.6(内存4GB，硬盘40GB，双核处理器)

开发与调试工具：bochs 2.6.8

## 实验步骤

* 掌握进程相关数据结构的定义方法：

进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相关的TSS，以及数据结构的关系

* 掌握构造进程的关键技术：

初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动

* 进程的现场保护与切换，弄清楚需要哪些关键数据结构与步骤：

时钟中断与进程调度关系，现场保护与恢复机理，从ring0-->ring1的上下文切换方法，中断重入机理

# 实验过程分析

（详细记录实验过程，通过截图展示得到的结果。特别是对于实验中发生的故障和问题，要进行故障分析，说明故障排除的过程及方法。）

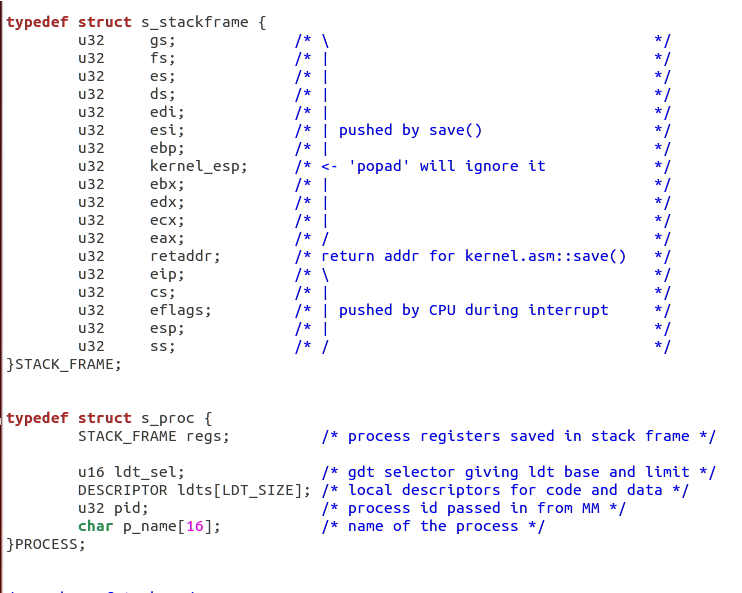
## 本次实验基本内容

### 掌握进程相关数据结构的定义方法：

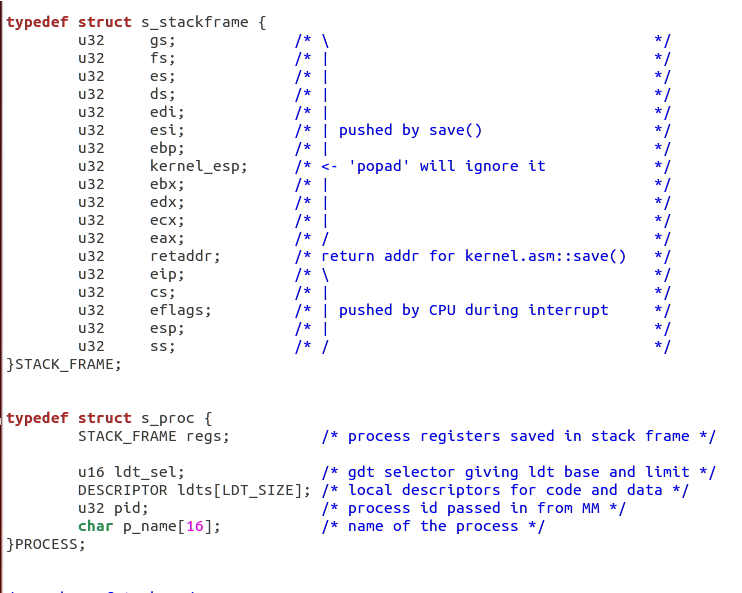
进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、 进程相关的TSS，以及数据结构的关系

1. **进程控制块(进程表)**

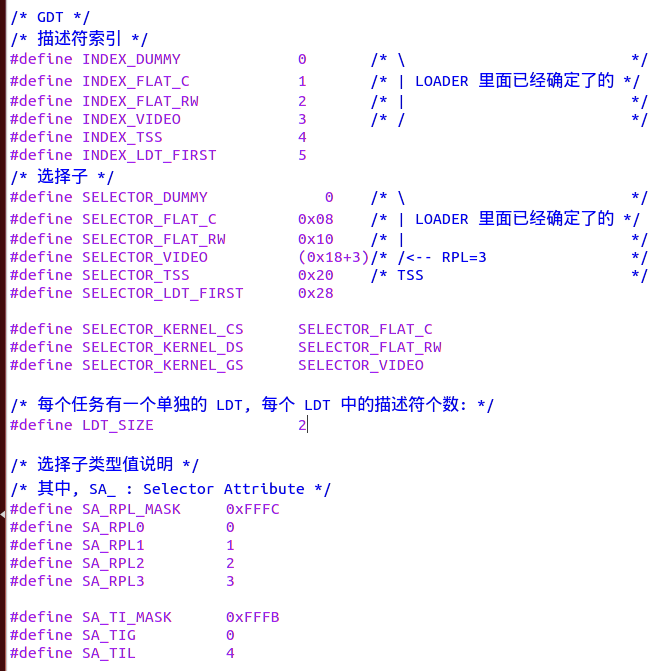
进程控制块PCB（Processing Control Block），是操作系统核心中一种数据结构，主要表示进程状态,通常是系统内存占用区中的一个连续存区，它存放着操作系统用于描述进程情况及控制进程运行所需的全部信息。



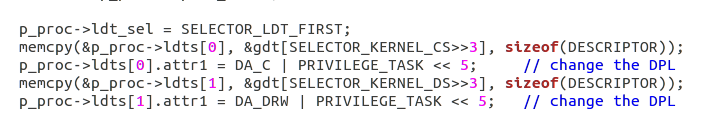
1. **进程结构体**



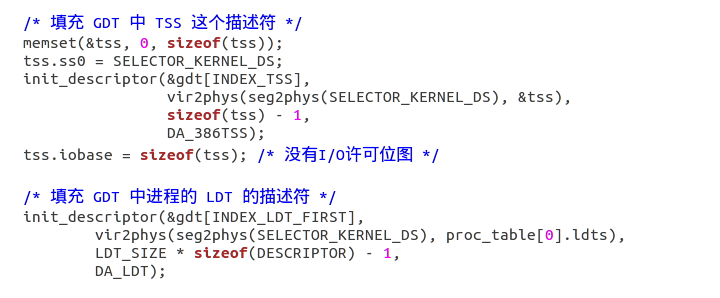
1. **进程相关的GDT/LDT**



GDT中定义了LDT的描述符和选择子，可以看到LDT里定义了两个描述符（LDT\_SIZE为2），之后会被分别初始化成内核代码段和内核数据段。如下图所示：

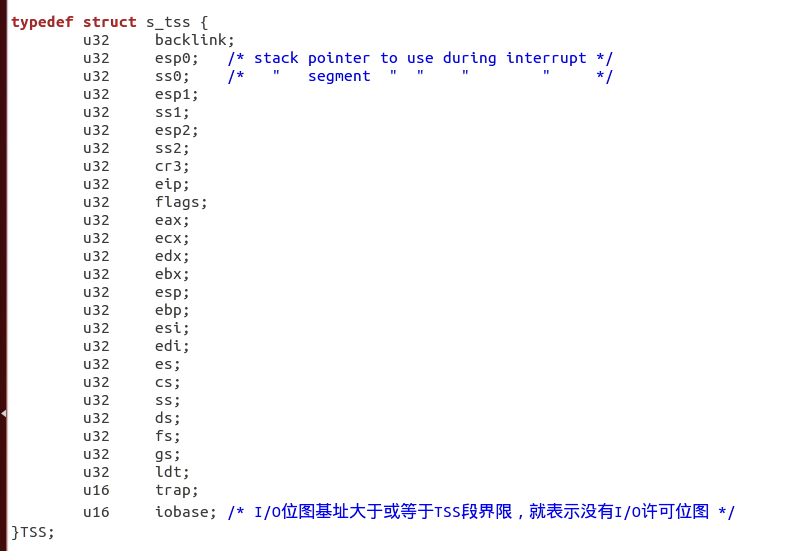


GDT还定义了TSS的描述符和选择子，在运行代码之前，需要填充GDT中进程的LDT和TSS的描述符。如下：



1. **进程相关的TSS**

TSS全称为task state segment，是指在操作系统进程管理的过程中，进程切换时的任务现场信息。与其他段一样，TSS也有描述它的结构：TSS描述符表，它记录了一个TSS的信息；同时还有一个TR寄存器，它指向当前任务的TSS，任务切换的时候，CPU会将原寄存器的内容写出到相应的TSS，同时将新TSS的内容填到各个寄存器中和新任务的TSS选择符更新到TR中，这样就实现了任务的切换。



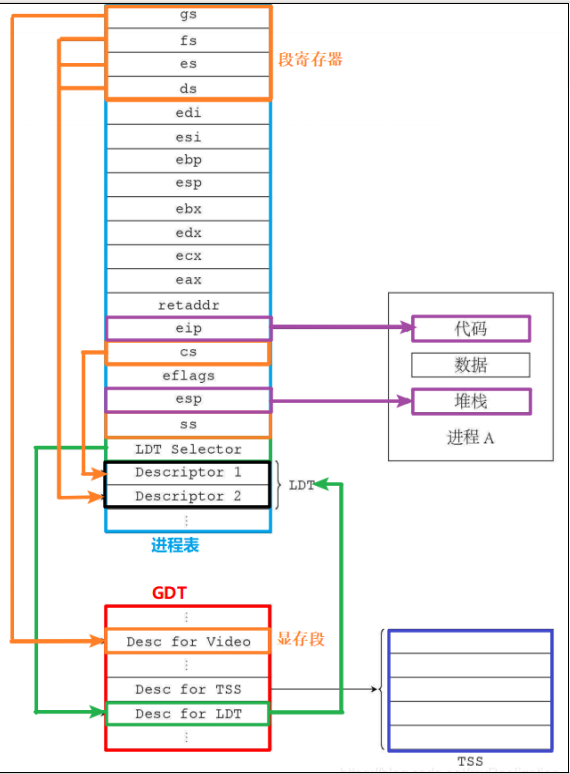
1. **数据结构的关系**

如下图所示，可以分为4个部分——进程表、进程体、GDT和TSS。关系如下：

·**进程表和GDT**——进程表内的LDT Selector对应GDT中的一个描述符，而这个描述符所指向的内存空间就存在于进程表内。（绿色线条部分）

·**进程表和进程**——进程表是进程的描述，进程运行过程中如果被中断，各个寄存器的值都会被保存进进程表中。但是，在我们的第一一个进程开始之前，并不需要初始化太多内容，只需要知道进程的入口地址就足够了。另外，由于程序免不了用到堆栈，而堆栈是不受程序本身控制的，所以还需要事先指定esp。（紫色线条部分）

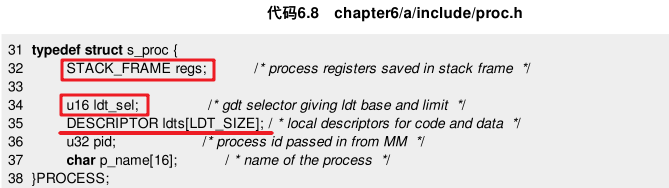
·**GDT和TSS**—— GDT中需要有一一个描述符来对应TSS,需要事先初始化这个描述符。（深蓝色线条部分）

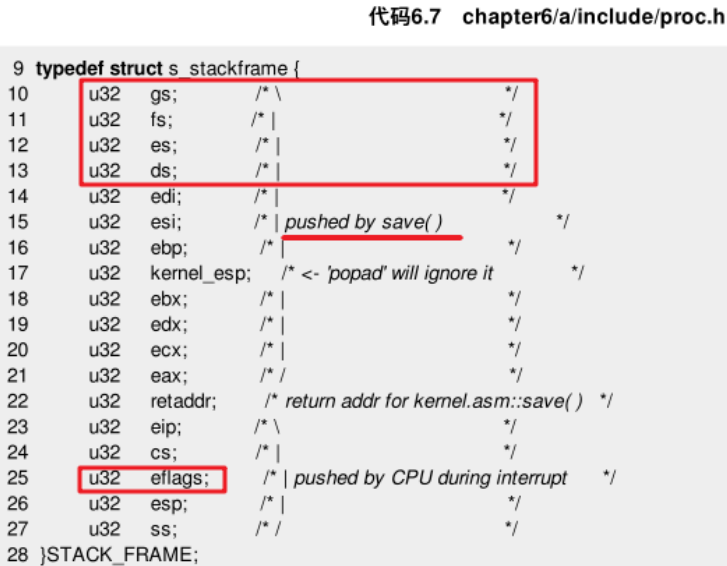


### 掌握构造进程的关键技术：

初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动。

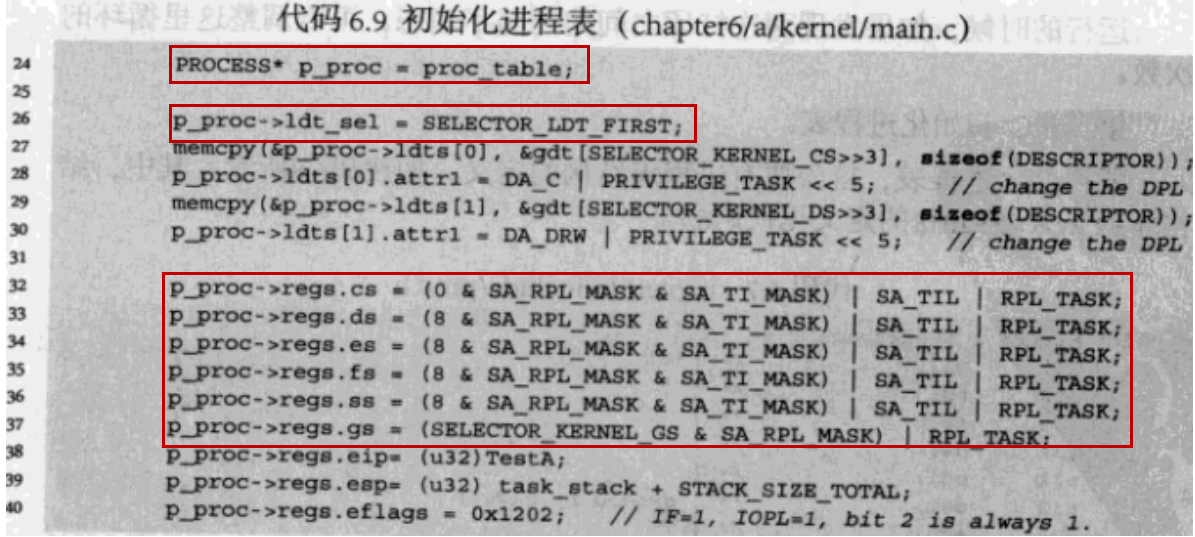
1. **初始化进程控制块的过程**
2. 初始化进程表，首先要有进程表结构 s\_proc 和结构体 STACK\_FRAME 的定义：



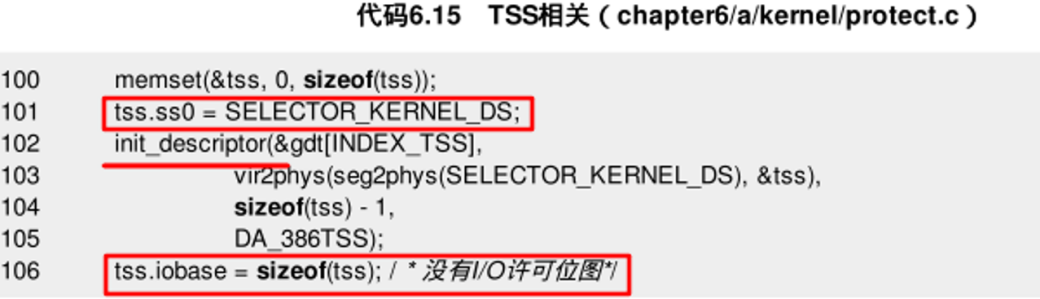


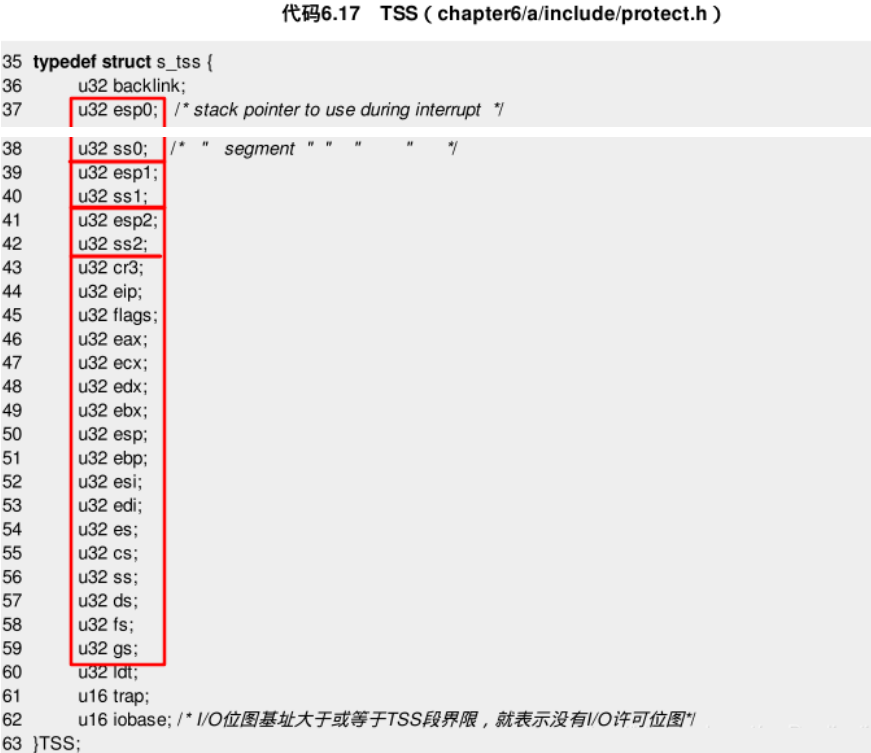
1. 在 global.c中 声明一个进程表：PUBLIC PROCESS proc\_table[NR\_TASKS]
2. 初始化进程表。kernel\_main()是最后一部分被执行的代码，因此初始化进程表的代码应添加在这里，进程表需要初始化的有3个部分：寄存器、LDT Selector 和LDT：

* LDT Selector 被赋值为SELECTOR\_LDT\_FIRST,这个宏的定义在代码protect.h中；
* LDT里面共有两个描述符，为简化起见，分别被初始化成内核代码段和内核数据段，只是改变了一下 DPL 以让其运行在低的特权级下；
* 要初始化的寄存器比较多，cs 指向LDT中第一个描述符，ds, es, fs, ss 都设为指向LDT中的第二个描述符，gs 仍然指向显存，只是其 RPL 发生改变。
* 接下来，eip 指向 TestA ，这表明进程将从 TestA 的入口地址开始运行；
* esp 指向了单独的栈，栈的大小为 STACK\_SIZE\_TOTAL ；
* 最后一行设置 eflags ，结合图3.46可以知道，0x1202 恰好设置了 IF 位并把 IOPL 设为 1 1 1 。这样，进程可以使用 I/O 指令，并且中断会在 iretd 执行时被打开：



1. 初始化GDT和TSS
2. 准备GDT和TSS。在 init\_prot( ) 中填充TSS以及对应的描述符：

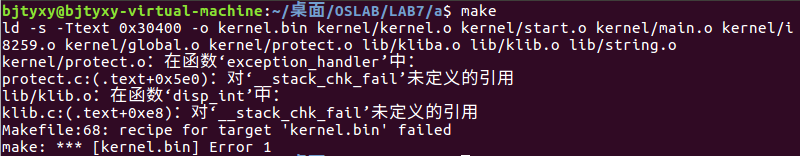




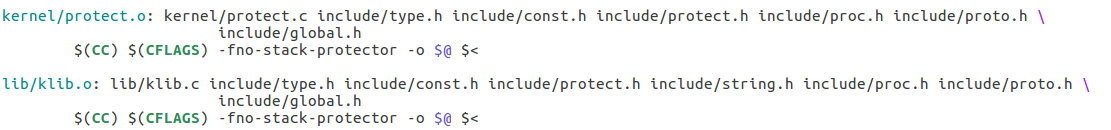
1. TSS准备好后，只要在 kernel.asm 中添加加载 tr 的代码即可：



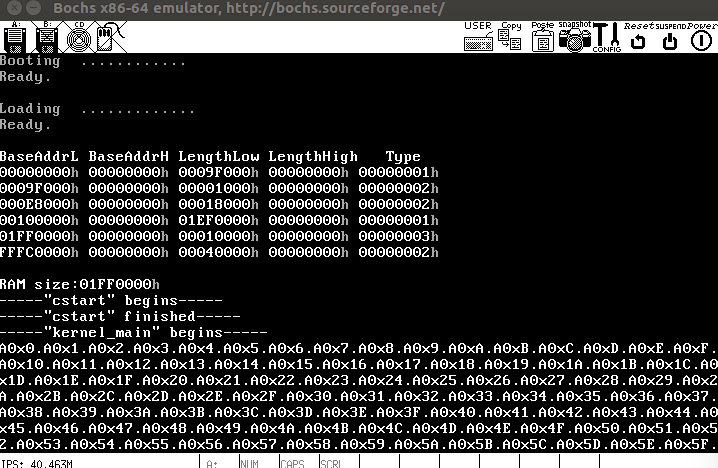
1. 实现进程的启动（参考代码 chapter6/a ）
2. make 并运行，发现报错



1. 对makefile文件做以下修改：添加-fno-stack-protector，取消栈保护机制



1. make后进入bochs运行，我们看到不断出现的字符 A 和不断增加的数字。这意味着 ring0 到 ring1 的跳转成功了，我们的进程在运行。



### 进程的现场保护与切换，弄清楚需要哪些关键数据结构与步骤

(时钟中断与进程调度关系，现场保护与恢复机理，从ring0-->ring1的上下文切换方法，中断重入机理)

时钟中断与进程调度关系

只有可能被改变的才有保存的必要,而进程要运行需要CPU和内存相互协作，且不同进程共用一个CPU的一套寄存器。所以，我们要把寄存器的值统统保存起来，在进程被恢复执行时使用。为了保证进程状态完整、不被破坏，要在进程刚刚被挂起时保存所有寄存器的值。我们通过使用push或者pushad指令将所有寄存器的值压入PCB，且这些代码应该写在时钟中断例程的最顶端，以便中断发生时马上被执行

现场保护与恢复机理

保存指令用的是push，将数据从PCB中恢复则使用pop指令。等所有寄存器的值都已经被恢复，再执行指令iretd ，就回到了那个进程

从ring0-->ring1的上下文切换方法

**Ring1->ring0:**如果由外层向内层转移时，需要从当前TSS中取出内层ss和esp作为目标代码的ss和esp。所以，我们必须事先准备好TSS 。

由于每个进程相对独立，我们把涉及到的描述符放在局部描述符表LDT中，所以，我们还需要为每个进程准备 LDT

**Ring0->ring1:** 刚才的分析过程中，我们假设初始状态是“进程A运行中”。可是到目前为止我们的代码完全运行在ring0。所以，当我们准备开始第一个进程时，我们面临一个从ring0到ring1的转移，并启动进程A。这跟我们从进程B恢复的情形很相似，完全可以在准备就绪之后跳转到中断处理程序的后半部分，假装发生了一次时钟中断来启动进程A，利用iretd来实现ring0到ring1的转移

## 本次实验要解决的问题

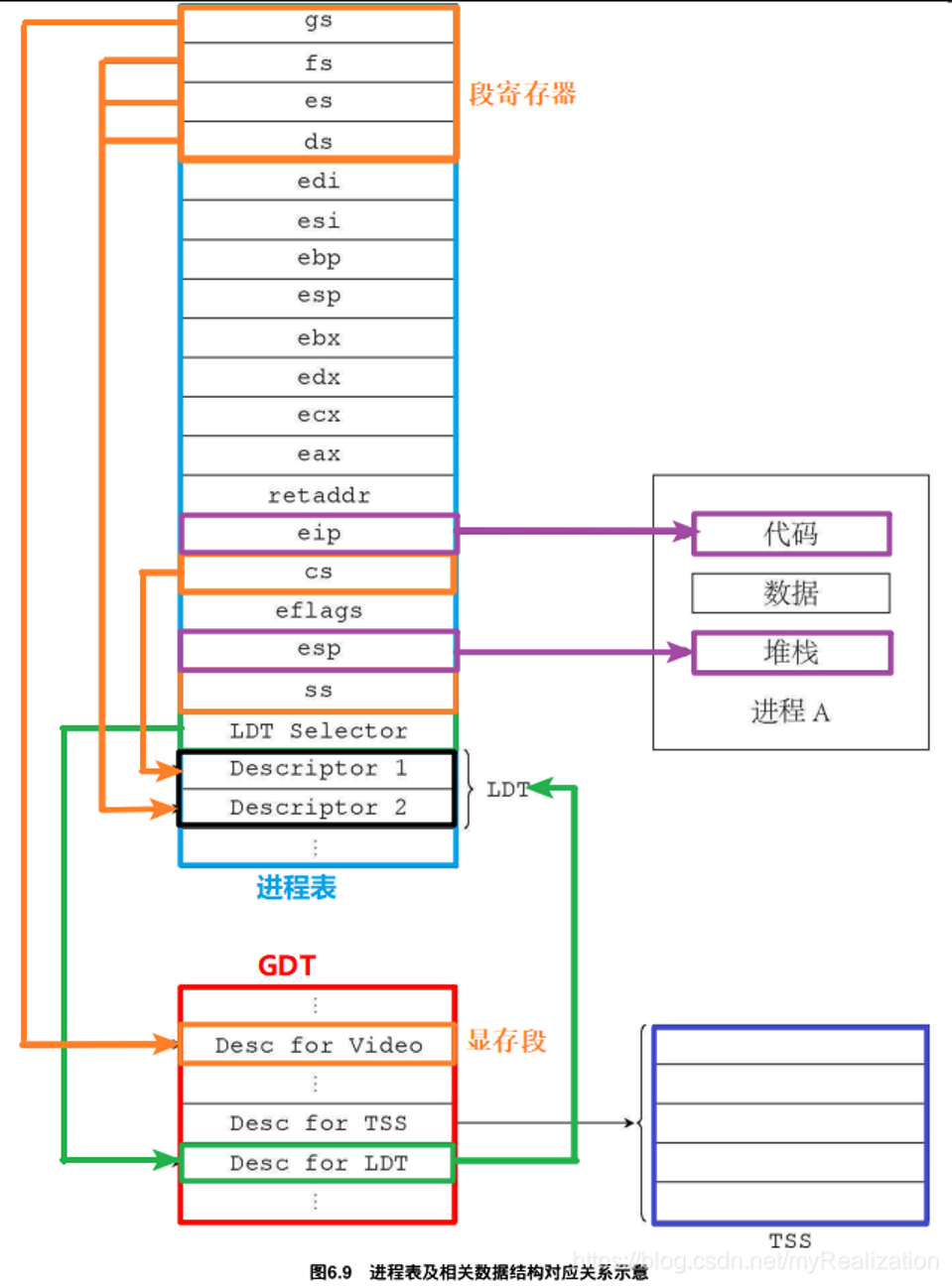
### 描述进程数据结构的定义与含义

进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相 关的TSS，画出数据结构的关系图。

1. 进程控制块/进程表（Process Control Block，PCB）：用于管理进程的数据结构，包含了进程的各种信息，如进程状态、程序计数器、堆栈指针、寄存器值等。进程控制块/进程表结构如下：

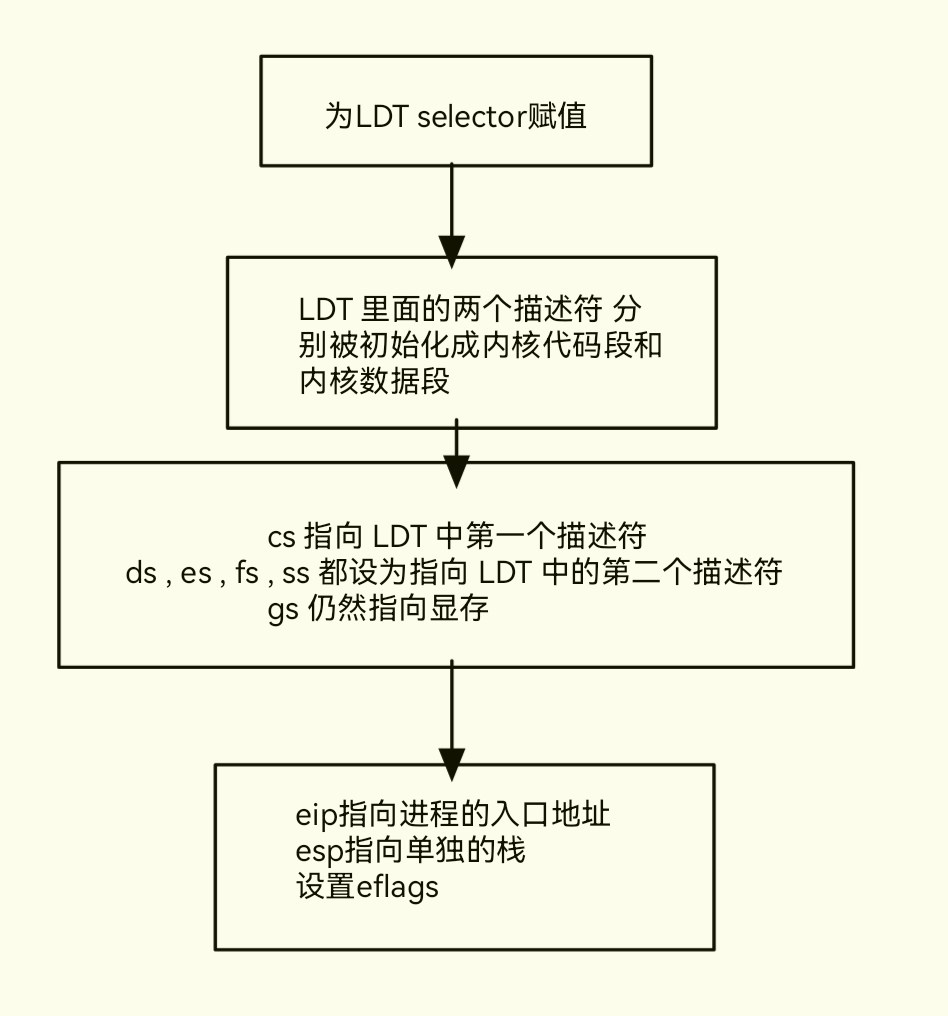


1. 进程结构体：C 语言中用于表示进程的数据结构，包含了进程的各种信息，如进程 ID、进程状态、进程优先级等。进程结构体通常会包含一个指向 PCB 的指针，以便操作系统可以访问和管理 PCB。
2. 进程相关的 GDT/LDT：用于存储全局描述符和局部描述符，包括代码段和数据段的描述符。每个进程都有一个单独的 GDT/LDT，用于描述进程的代码和数据段。
3. 进程相关的 TSS：用于存储任务的状态信息，如任务的堆栈指针、任务的 I/O 权限等。每个进程都有一个单独的 TSS，用于描述进程的状态信息。
4. 数据结构关系图如下：

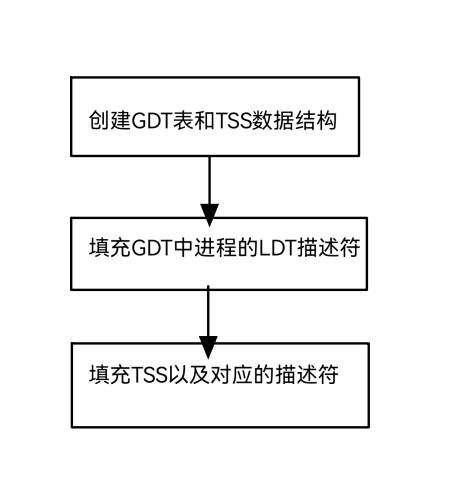


### 画出以下关键技术的流程图：

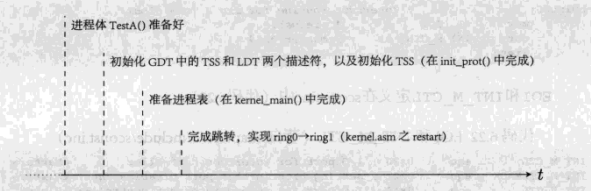
初始化进程控制块的过程图：



初始化GDT和TSS的流程图：



进程的启动：



### 怎么实现进程的现场保护与恢复？

现场保护和现场恢复是利用堆栈和堆栈指令 PUSH 和 POP 来实现的。现场保护时，是将各个寄存器的内容进栈;现场恢复时，是将各个寄存器的内容出栈。

结合TSS来说实现进程的现场保护与恢复可以通过以下步骤：

现场保护：

（1）创建TSS：操作系统在内存中为每个任务创建一个TSS，并分配空间以保存任务的上下文信息。

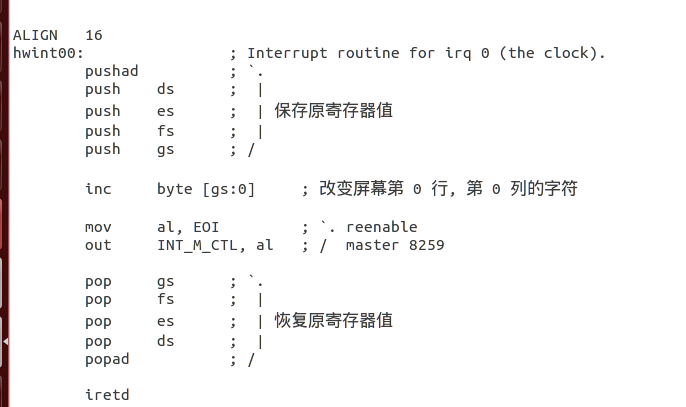
（2）保存进程上下文到TSS：当进行进程切换时，操作系统将当前进程的寄存器状态、堆栈指针以及其他必要信息保存到该进程对应的TSS中。寄存器状态通过指令TASK SWITCH或者HLT等指令来触发TSS的更新，将当前寄存器状态保存到TSS中。

现场恢复：

（1）加载TSS：当需要恢复某个进程时，操作系统将相应任务的TSS加载到处理器中。

（2）从TSS中恢复进程上下文：处理器会根据加载的TSS来更新寄存器状态，将保存在TSS中的寄存器值加载到处理器对应的寄存器中。同样，堆栈指针和其他必要的状态也会从TSS中加载并恢复。

在本实验中现场保护的过程就是进程信息压栈，将寄存器的值进行压栈，然后要将esp设置为TSS中预设的值，之后中断会发生要将esp指向内核栈。恢复的过程就是将push的过程替换为pop，并将esp的值重新设置。

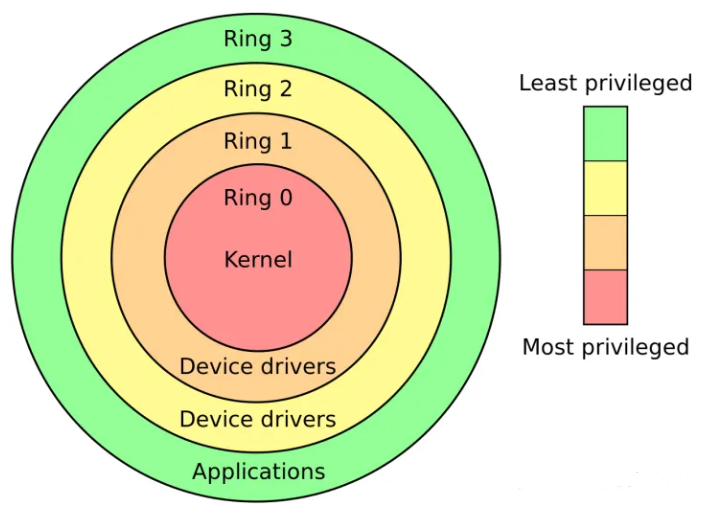


### 为什么需要从ring0-->ring1，怎么实现？

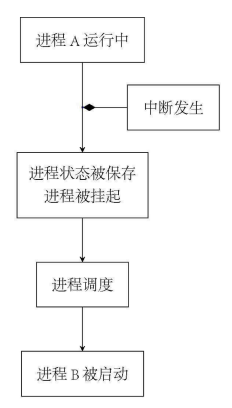
我们首先介绍ring0和ring1的概念。CPU将特权级别分为4个级别：ring0、ring1、ring2、ring3，其特权级别依次降低。

操作系统的代码在最高级别ring0上运行，可以使用特权指令、控制中断、修改页表等功能；应用程序的代码在最低级别ring3上运行，权限很低，不能访问ring0时的地址空间。如果应用程序需要执行较高特权的行为，如访问磁盘等，需要执行系统调用。执行系统调用的时候，CPU的运行级别会从ring3切换到ring1，并跳转到系统调用对应的内核代码，然后继续执行特权功能，执行的内核代码将使用当前进程的内核栈。此时，程序会从用户态转换为内核态。执行完毕，CPU再从ring0返回到ring3，程序也从内核态回到用户态。

Linux系统只使用到ring0和ring3两种状态。



当正在执行的用户程序被运行在ring0层的中断程序中断时，此时程序也处于进程的内核态。回到本题，本实验中的时钟中断程序运行在ring0层，而进程A运行在ring1层，所以CPU需要从ring1**-->**ring0；中断处理程序完成，下一个应该运行在ring1层上的进程B被指定，所以CPU需要从ring0**-->**ring1。



使用iretd指令可以实现从ring0到ring1的转移。然而，用户态的进程需要首先从ring1转换到ring0，才能实现以上操作。本实验假设有一个时钟中断将该进程陷入到内核态，此时，当前用户态进程的上下文被压入TSS指定的堆栈，然后切换到内核栈，执行中断处理程序。返回时，系统首先初始化进程表的寄存器、LDT、LDT Selector、TSS等信息，然后跳转到restart，使用lldt命令设置ESP的值，跳过retaddr，使用iretd返回到用户态，也就是返回到ring1。

restart具体代码如下：

restart:

mov esp, [p\_proc\_ready]

lldt [esp + P\_LDT\_SEL]

lea eax, [esp + P\_STACKTOP]

mov dword [tss + TSS3\_S\_SP0], eax

pop gs

pop fs

pop es

pop ds

popad

add esp, 4

iretd

### 什么是中断重入，具体怎么实现，画出流程图？

在进程切换中，可能会出现如下情况：一次中断还没有处理完，又一次中断发生了，此时程序又跳转到中断程序的开头，反复下去，使得程序难以跳回到进程继续执行。并且，每一次中断都意味着压栈操作，不断的压栈会导致堆栈溢出。为了防止这种情况，需要设置一个全局变量，记录目前已经开始但没有结束的中断处理程序的数量。该全局变量k\_reenter的初值为-1，当一个中断处理程序开始时，该值加1；一个中断处理程序结束时，该值减1。

那么，中断处理程序开始时可以检查k\_reenter的值，当其不为0，也就表示此时还存在没有完成的中断处理程序时，可以直接跳到最后，结束新的中断。

**inc dword [k\_reenter] ; 中断开始，先增加k\_reenter的值**

**cmp dword [k\_reenter], 0 ; 检查是否还有其他未完成的中断处理程序**

**jne .re\_enter ; 如果有，跳转到re\_enter，直接结束程序**

**...**

**.re\_enter: ; 如果(k\_reenter != 0)，会跳转到这里**

**dec dword [k\_reenter]**

**pop gs**

**pop fs**

**pop es ; 恢复原寄存器值**

**pop ds**

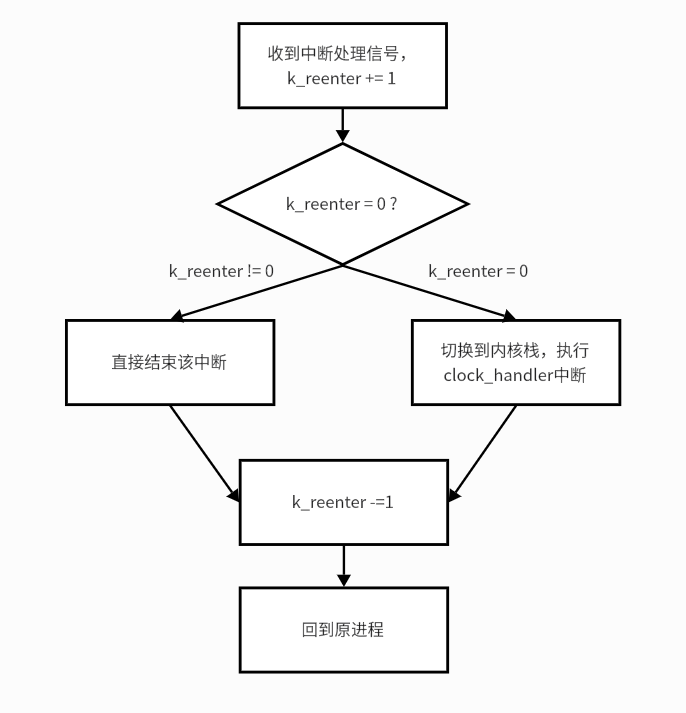
**popad**

**add esp, 4**

**iretd**

而在课本的6.4节中，程序进行了改进，之前的程序在发生中断重入时不执行clock\_handler，但改进之后的程序可以始终执行clock\_handler。具体来说，程序操纵8259A，避免在处理当前中断的同时发生同样类型的中断，之后给8259A寄存器发出中断结束命令。这样，可以避免在clock\_handler执行的过程中出现时钟中断的中断重入，但是它仍然不能阻止其他类型的中断。

中断重入的流程图如下所示：



# 实验结果总结

（对实验结果进行分析。并理论联系实际，思考并列出本实验对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。）

## 进程

### 进程的定义

从理论角度看，是对正在运行的程序过程的抽象；从实现角度看，是一种数据结构，目的在于清晰地刻画动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。

### 进程的特征

• 动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态消亡的。

• 并发性：任何进程都可以同其他进程一起并发执行

• 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位

• 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。

### 进程的状态

• 就绪状态：进程已获得除处理器外的所需资源，等待分配处理器资源；只要分配了处理器进程就可执行。就绪进程可以按多个优先级来划分队列。例如，当一个进程由于时间片用完而进入就绪状态时，排入低优先级队列；当进程由 I/O 操作完成而进入就绪状态时，排入高优先级队列。

• 运行状态：进程占用处理器资源；处于此状态的进程的数目小于等于处理器的数目。在没有其他进程可以执行时 (如所有进程都在阻塞状态)，通常会自动执行系统的空闲进程。

• 阻塞状态：由于进程等待某种条件（如 I/O 操作或进程同步），在条件满足之前无法继续执行。该事件发生前即使把处理机分配给该进程，也无法运行。

## 进程表

内核负责管理维护所有进程，为了管理进程，内核在内核空间维护了一个称为进程表（Process Table）的数据结构，这个数据结构中记录了所有进程，每个进程在数据结构中都称为一个进程表项（Process Table Entry）。进程表在进行上下文切换时，能够保存下在 CPU 中关于当前运行进程的一些重要寄存器信息。

因为在进程切换的过程中，esp 会分别指向不同的位置，如果入出栈操作不当容易造成数据被破坏，因此除了进程表，还要引入一个内核栈，esp 的位置出现在这3个不同区域：

• 进程栈–进程运行时自身的堆栈

• 进程表–存储进程状态信息的数据结构

• 内核栈–进程调度模块运行时使用的堆栈

## 进程的切换，从ring0->ring1

从 ring0 切换到 ring1，只需要初始化进程表的寄存器、LDT Selector 和 LDT，再初始化 TSS 和对应的描述符，最后使用 restart() 完成跳转即可。

从 ring1 切换到 ring0，触发的条件一个是时钟中断，每次发生中断的时候，此时的堆栈是进程表的堆栈，由硬件自动把 ss,esp,eflag,cs,ip 压入 tss 制定的堆栈中，然后执行到中断处理程序时压入一个retaddr，接着压入其他的寄存器。然后切换到内核栈，执行中断处理程序，里面有进程调度程序，就是p\_proc\_ready 指向不同的进程表，之后赋值给 esp，此时堆栈又切换为进程表的堆栈，依次 pop 之后，iretd, 进程切换到 ring0，此时的堆栈是进程的堆栈。

# 个人分工及心得体会

（每个人分别填写自己在本次实验中的分工，并总结实验的心得体会。）

## 张子航

完成本次实验内容的第3题，以及本次实验要解决的第2题，并撰写实验报告中的对应部分及实验结果总结部分

通过本次实验，我掌握到了进程相关数据结构的定义方法，了解到了进程控制块(进程表)、进程相关的GDT/LDT、进程相关的TSS的结构和关系。同时，通过本次实验我还学习到了初始化进程控制块的过程、初始化GDT和TSS、实现进程的启动以及进程的现场保护与切换等步骤的实现机理。此外，本次实验还加深了我对于操作系统课程中的时钟中断、进程调度等知识的理解，我进行后续的实验打下了基础。

## 辜汝曦

完成本次实验的全部内容，并撰写实验报告中本次实验要解决的问题的第4题、第5题，以及实验目的及实验内容部分。

在本次实验中，我初步了解了进程的实现机理与进程管理部分。之前在操作系统理论课上，我只是对进程的用户态、内核态有一个大致的了解，在本次实验中，我探究了程序是如何在用户态与内核态进行切换的，认识了TSS等进程相关的数据结构。我也实践了程序从ring0到ring1的转换，为之后的多进程切换实验打下基础。

## 杨馨悦

完成本次实验内容第2题，本次实验要解决的问题的第1题，并撰写实验报告的对应部分及实验结果总结部分。

通过本次实验，我对进程的实现机理与进程管理的相关知识有了总体的了解。我掌握了进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、 进程相关的TSS的概念以及它们之间的数据结构关系，熟悉了初始化进程控制块的过程、初始化GDT和TSS和进程启动这三个构造进程的关键技术，了解了进程切换、现场保护与恢复机理，同时还了解了环保护、进程调度与CPU模式切换等相关知识。

## 赵敏

完成本次实验内容第1题，本次实验要解决的问题的第3题，并撰写实验报告的对应部分。

在本次实验中，我们终于接触到了在理论课中熟悉的进程部分，但是呈现在代码中却显得有些晦涩难懂，通过反复阅读，逐渐了解到进程数据结构的定义与含义如进程控制块(进程表)、进程结构体、进程相关的GDT/LDT、进程相关的TSS等，并分析了它们的关系，学习到一个简单的进程是如何实现的，同时还学习到进程切换过程中现场的保护与恢复方式，ring1到ring0的切换等等。