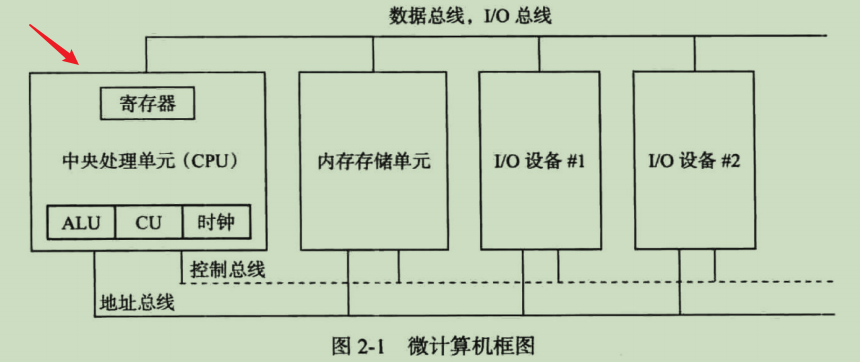
# CPU

## 1.1 CPU架构

CPU架构是CPU厂商给属于同一系列的CPU产品定的一个规范，主要目的是为了区分不同类型CPU的重要标示。

目前市面上的CPU指令集分类主要分有两大阵营，一个是intel、AMD为首的复杂指令集CPU，另一个是以IBM、ARM为首的精简指令集CPU。两个不同品牌的CPU，其产品的架构也不相同，例如，Intel、AMD的CPU是X86架构的，而IBM公司的CPU是PowerPC架构，ARM公司是ARM架构。

CPU(中央处理器)是进行算术和逻辑操作的部件，包含了有限数量的寄存器、一个高频时钟、一个控制单元和一个算术逻辑单元。



* 时钟：用一个恒定速率的内部时钟脉冲对cpu内部操作与系统其他组件进行同步；机器指令的基本时间单位是时钟周期或机器周期；
* 时钟周期：

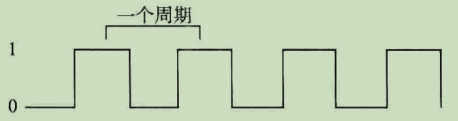
也称作振荡周期(晶振周期)、节拍脉冲或T周期，等于时钟频率的倒数，时钟周期是计算机中最基本的、最小的时间单位。

在一个时钟周期内，CPU仅完成一个最基本的动作。

一个时钟周期的时长和一个完整的时钟脉冲的时间相同，时钟周期持续时间等于时钟速度(CPU每秒振荡的次数,如1GHz)的倒数(即 1/时钟速度)，例如一个每秒振荡10亿次(1GHz)的时钟，其时钟周期为10亿分之一秒(即1纳秒)；执行一条机器指令至少需要一个时钟周期, 有的需要多个时钟周期的时间(比如乘法指令)；

一条cpu指令执行时间需要n(n≥1)个时钟周期；

cpu访问一次寄存器一般只需要1个时钟周期；



* 指令执行周期：

cpu执行一条指令(取址、译码和执行)的时间；

* 控制单元(CU)：协调机器指令执行的步骤序列；
* 算术逻辑单元(ALU)：执行算术运算和逻辑运算；
* 总线：
* cpu通过自身引脚和主板cpu插槽和计算机其他部分相连；所有的数据处理都在cpu内部进行，因此内存中的程序及数据在执行之前需要被复制到cpu中，cpu在复制程序指令时可以一次复制一条或多条；当cpu执行程序时，会预先把后续的一千条指令存入cpu内部的高速缓存中

## 1.2 微处理器及CPU的发展历程

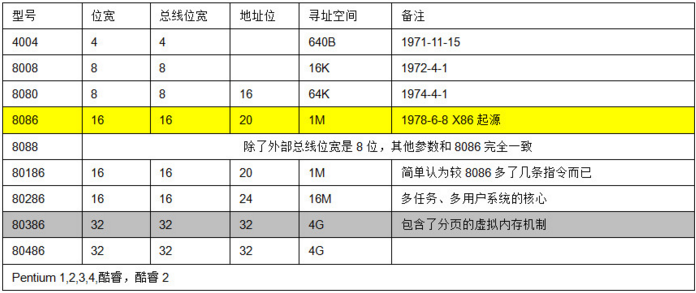
1947年，美国贝尔实验室的肖克利和同事一起发明了晶体管。

1958~1959年，美国人杰克·基尔比发明了锗集成电路，罗伯特·诺伊斯(Intel公司创始人)发明了硅集成电路。

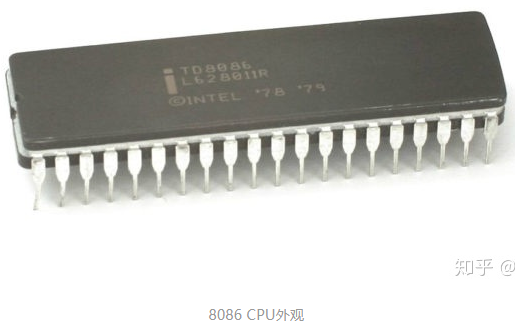
1968年7月18日，鲍勃-诺斯和戈登-摩尔的新公司在美国加利福尼亚州，美丽的圣弗朗西斯科湾畔芒延维尤城的梅多费大街365号开张了。并在成立不久斥资15000美元从一家叫INTELCO的公司手中买下了Intel名称的使用权创立Intel公司。由此Intel这位半导体巨人开始了他在IT行业传奇般的历史。

1971年11月15日，这一天被当作全球IT界具有里程碑意义的日子而被写入许多计算机专业教科书。Intel公司的工程师特德·霍夫发明了世界上第一个商用微处理器—4004，这款4位微处理器虽然只有45条指令，而且每秒只能执行5万条指令。甚至比不上1946年由美国陆军宾夕法尼亚大学研制的世界第一台计算机ENIAC。但它的集成度却要高很多，一块4004的重量还不到一盅司。 他因发明了微处理器，被英国《经济学家》杂志称为“第二次世界大战以来最有影响的科学家之一”。





Intel从8086开始，就进入了我们所谓的x86时代。而80386的诞生则标志着Intel正是进入了32位微处理器的时代。从80386到Pentium 4这个年代的CPU，就是传说中的IA-32时代。



## 1.3 微处理器概念

微处理器是由一片或少数几片大规模集成电路组成的中央处理器，这些电路执行算术运算和控制功能。微处理器能完成取指令、执行指令，以及与外界存储器和逻辑部件交换信息等操作，是微型计算机的运算控制部分。它可与存储器和外围电路芯片组成微型计算机。微处理器与传统的中央处理器相比，具有体积小、重量轻和容易模块化等优点。微处理器的基本组成部分有：寄存器堆、运算器、时序控制电路，以及数据和地址总线。

微处理器是微型处理器的统称！ 比如CPU、显卡的GPU、手机的处理器，以及一些智能家电的处理器。所谓处理器就是一个执行所有计算功能的部件。

自从人类1947年发明晶体管以来，50多年间半导体技术经历了硅晶体管、集成电路、超大规模集成电路、甚大规模集成电路等几代，发展速度之快是其他产业所没有的。半导体技术对整个社会产生了广泛的影响，因此被称为“产业的种子”。中央处理器是指计算机内部对数据进行处理并对处理过程进行控制的部件，伴随着大规模集成电路技术的迅速发展，芯片集成密度越来越高，CPU可以集成在一个半导体芯片上，这种具有中央处理器功能的大规模集成电路器件，被统称为“微处理器”。需要注意的是：微处理器本身并不等于微型计算机，仅仅是微型计算机的中央处理器。

微处理器已经无处不在，无论是录像机、智能洗衣机、移动电话等家电产品，还是汽车引擎控制，以及数控机床、导弹精确制导等都要嵌入各类不同的微处理器。微处理器不仅是微型计算机的核心部件，也是各种数字化智能设备的关键部件。国际上的超高速巨型计算机、大型计算机等高端计算系统也都采用大量的通用高性能微处理器建造。

根据微处理器的应用领域，微处理器大致可以分为三类：通用高性能微处理器、嵌入式微处理器和数字信号处理器、微控制器。一般而言，通用处理器追求高性能，它们用于运行通用软件，配备完备、复杂的操作系统；嵌入式微处理器强调处理特定应用问题的高性能，主要用于运行面向特定领域的专用程序，配备轻量级操作系统，主要用于蜂窝电话、CD播放机等消费类家电；微控制器价位相对较低，在微处理器市场上需求量最大，主要用于汽车、空调、自动机械等领域的自控设备。

## x86处理器架构

## 1.5 寄存器

### 1.5.1 概念

内存和寄存器就是为了解决存储器读写速度慢而产生的多级存储机制。从20世纪50年代开始，磁芯存储器曾一度成为主存的主要存储介质，但从20世纪70年代开始，逐步被半导体存储器所取代，目前的计算机都是用半导体存储器。

寄存器是中央处理器内的组成部分。寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址。在中央处理器的控制部件中，包含的寄存器有指令寄存器(IR)和程序计数器(PC)。在中央处理器的算术及逻辑部件中，寄存器有累加器(ACC)。

在计算机领域，寄存器是CPU内部的元件，包括通用寄存器、专用寄存器和控制寄存器。寄存器拥有非常高的读写速度，所以在寄存器之间的数据传送非常快。

### 1.5.2 分类

* 数据寄存器- 用来储存整数数字（参考以下的浮点寄存器）。在某些简单/旧的CPU，特别的数据寄存器是累加器，作为数学计算之用。
* 地址寄存器- 持有存储器地址，用来访问存储器。在某些简单/旧的CPU里，特别的地址寄存器是索引寄存器（可能出现一个或多个）。
* 通用目的寄存器（GPRs） - 可以保存数据或地址两者，也就是说它们是结合数据/地址 寄存器的功用。
* 浮点寄存器（FPRs） - 用来储存浮点数字。
* 常数寄存器- 用来持有只读的数值（例如0、1、圆周率等等）。
* 向量寄存器- 用来储存由向量处理器运行SIMD（Single Instruction, Multiple Data）指令所得到的数据。
* 特殊目的寄存器- 储存CPU内部的数据，像是程序计数器（或称为指令指针），堆栈寄存器，以及状态寄存器（或称微处理器状态字组）。
* 指令寄存器（instruction register）- 储存现在正在被运行的指令。
* 索引寄存器（index register）- 是在程序运行时用来更改运算对象地址之用。

在某些架构下，模式指示寄存器（也称为“机器指示寄存器”）储存和设置跟处理器自己有关的数据。由于他

PORT1的控制寄存器

PORT1的控制寄存器

们的意图目的是附加到特定处理器的设计，因此他们并不被预期会成为微处理器世代之间保留的标准。

有关从随机存取存储器提取信息的寄存器与CPU（位于不同芯片的储存寄存器集合）

* 存储器缓冲寄存器（Memory buffer register）
* 存储器数据寄存器（Memory data register）
* 存储器地址寄存器（Memory address register）
* 存储器型态范围寄存器（Memory Type Range Registers）[1]
* 向量寄存器

## 4.3 8086CPU中的寄存器

### 4.3.1 通用寄存器

有8个, 又可以分成2组，一组是数据寄存器(4个),另一组是指针寄存器及变址寄存器(4个)；

通用寄存器是那些你可以根据自己的意愿使用的寄存器，修改他们的值通常不会对计算机的运行造成很大的影响。

#### 4.3.1.1 数据寄存器分为

他们的特点是，这4个16位的寄存器可以分为高8位: AH, BH, CH, DH.以及低八位：AL,BL,CL,DL。这2组8位寄存器可以分别寻址，并单独使用。

1. AH&AL=AX(accumulator)：累加寄存器，常用于运算;在乘除等指令中指定用来存放操作数，另外,所有的I/O指令都使用这一寄存器与外界设备传送数据。
2. BH&BL=BX(base)：基址寄存器，常用于地址索引
3. CH&CL=CX(count)：计数寄存器，常用于计数；常用于保存计算值，如在移位指令,循环(loop)和串处理指令中用作隐含的计数器.
4. DH&DL=DX(data)：数据寄存器，常用于数据传递

#### 4.3.1.2 指针寄存器和变址寄存器

这4个16位寄存器只能按16位进行存取操作，主要用来形成操作数的地址，用于堆栈操作和变址运算中计算操作数的有效地址。

SP（Stack Pointer）：堆栈指针，与SS配合使用，可指向目前的堆栈位置

BP（Base Pointer）：基址指针寄存器，可用作SS的一个相对基址位置

SI（Source Index）：源变址寄存器，可用来存放相对于DS段之源变址指针

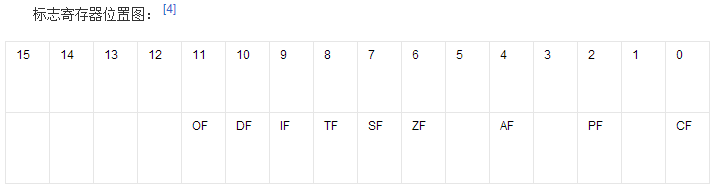
DI（Destination Index）：目的变址寄存器，可用来存放相对于ES 段之目的变址指针。

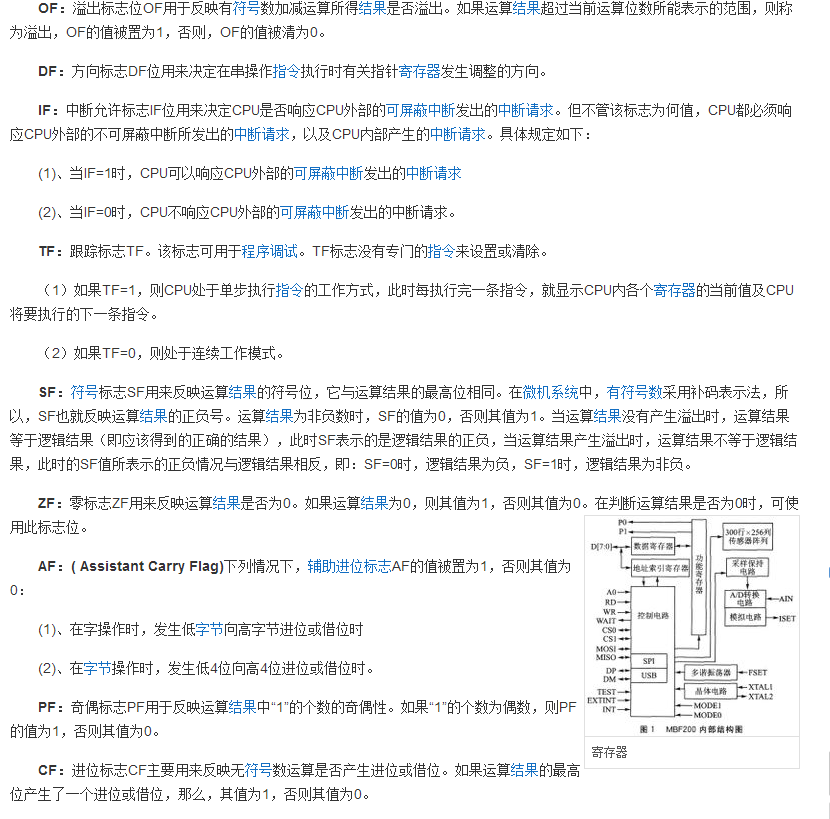
### 4.3.2 指令指针IP

指令指针IP是一个16位专用寄存器，它指向当前需要取出的指令字节，当BIU从内存中取出一个指令字节后，IP就自动加(取出该字节的长度，如：BIU从内存中取出的是1个字节，IP就会自动加1，如果BIU从内存中取出的字节数长度为3，IP就自动加3)，指向下一个指令字节。注意，IP指向的是指令地址的段内地址偏移量，又称偏移地址(Offset Address)或有效地址(EA，Effective Address)。

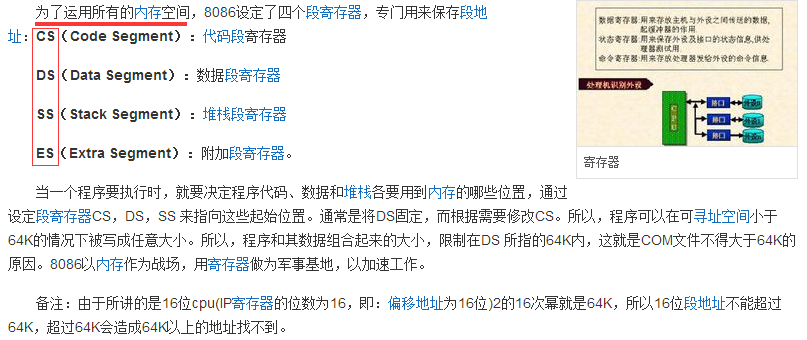
### 4.3.3 标志寄存器

8086有一个16位的标志性寄存器FR，在FR中有意义的有9位，其中6位是状态位，3位是控制位。标志寄存器（Flags Register,FR）又称程序状态字(Program Status Word,PSW)。这是一个存放条件标志、控制标志寄存器，主要用于反映处理器的状态和运算结果的某些特征及控制指令的执行。





### 4.3.4 段寄存器



# CPU的工作模式

## 演化历程

在微处理器的历史上，第一款微处理器芯片4004是由Intel推出的，那是一个4位的微处理器。在4004之后，intel推出了一款8位处理器8080，它有1个主累加器（寄存器A）和6个次累加器（寄存器B,C,D,E,H和L）,几个次累加器可以配对（如组成BC, DE或HL）用来访问16位的内存地址，也就是说8080可访问到64K内的地址空间。另外，那时还没有段的概念，访问内存都要通过绝对地址，因此程序中的地址必须进行硬编码（给出具体地址），而且也难以重定位，这就不难理解为什么当时的软件大都是些可控性弱,结构简陋，数据处理量小的工控程序了。

几年后，intel开发出了16位的处理器8086，这个处理器标志着Intel X86王朝的开始，这也是内存寻址的第一次飞跃。之所以说这是一次飞跃，是因为8086处理器引入了一个重要概念—段。

8086处理器的寻址目标是1M大的内存空间，于是它的地址总线扩展到了20位。但是，一个问题摆在了Intel设计人员面前，虽然地址总线宽度是20位的，但是CPU中“算术逻辑运算单元（ALU）”的宽度，即数据总线却只有16位，也就是可直接加以运算的指针长度是16位的。如何填补这个空隙呢？可能的解决方案有多种，例如，可以像一些8位CPU中那样，增设一些20位的指令专用于地址运算和操作，但是那样又会造成CPU内存结构的不均匀。又例如，当时的PDP－11小型机也是16位的，但是其内存管理单元（MMU）可以将16位的地址映射到24位的地址空间。受此启发，Intel设计了一种在当时看来不失为巧妙的方法，即分段的方法。

为了支持分段，Intel在8086 CPU中设置了四个段寄存器：CS、DS、SS和ES，分别用于可执行代码段、数据段、堆栈段及其他段。每个段寄存器都是16位的，对应于地址总线中的高16位。每条“访内”指令中的内部地址也都是16位的，但是在送上地址总线之前，CPU内部自动地把它与某个段寄存器中的内容相加。因为段寄存器中的内容对应于20位地址总线中的高16位(也就是把段寄存器左移4位)，所以相加时实际上是内存总线中的高12位与段寄存器中的16位相加，而低4位保留不变，这样就形成一个20位的实际地址，也就实现了从16位内存地址到20位实际地址的转换，或者叫“映射”。

段式内存管理带来了显而易见的优势，程序的地址不再需要硬编码了，调试错误也更容易定位了，更可贵的是支持更大的内存地址。程序员开始获得了自由。

技术的发展不会就此止步。intel的80286处理器于1982年问世了，它的地址总线位数增加到了24位，因此可以访问到16M的内存空间。更重要的是从此开始引进了一个全新理念—保护模式。这种模式下内存段的访问受到了限制。访问内存时不能直接从段寄存器中获得段的起始地址了，而需要经过额外转换和检查（从此你不能再随意存取数据段,具体保护和实现我们后面讲述）。

为了和过去兼容，80286内存寻址可以有两种方式，一种是先进的保护模式，另一种是老式的8086方式，被成为实模式。系统启动时处理器处于实模式，只能访问1M空间，经过处理可进入保护模式，访问空间扩大到16M，但是要想从保护模式返回到实模式，你只有重新启动机器。还有一个致命的缺陷是80286虽然扩大了访问空间，但是每个段的大小还是64k，程序规模仍受到限制。因此这个先天低能儿注定命不会很久。很快它就被天资卓越的兄弟——80386代替了。

80386是一个32位的CPU，也就是它的ALU数据总线是32位的，同时它的地址总线与数据总线宽度一致，也是32位，因此，其寻址能力达到4GB。对于内存来说，似乎是足够了。从理论上说，当数据总线与地址总线宽度一致时，其CPU结构应该简洁明了。但是，80386无法做到这一点。作为X86产品系列的一员，80386必须维持那些段寄存器的存在，还必须支持实模式，同时又要能支持保护模式，这给Intel的设计人员带来很大的挑战。

Intel选择了在段寄存器的基础上构筑保护模式，并且保留段寄存器16位。在保护模式下,它的段范围不再受限于64K，可以达到4G。这一下真正解放了软件工程师,他们不必再费尽心思去压缩程序规模，软件功能也因此迅速提升。

从8086的16位到80386的32位处理器，这看起来是处理器位数的变化，但实质上是处理器体系结构的变化，从寻址方式上说，就是从“实模式”到“保护模式”的变化。从80386以后，Intel的CPU经历了80486、Pentium、PentiumII、PentiumIII等型号，虽然它们在速度上提高了好几个数量级，功能上也有不少改进，但基本上属于同一种系统结构的改进与加强，而无本质的变化，所以我们把80386以后的处理器统称为IA32（32 Bit Intel Architecture）。

从80386开始，cpu有三种工作方式：实模式，保护模式和虚拟8086模式。只有在刚刚启动的时候是real-mode，等到linux操作系统运行起来以后就运行在保护模式。

## 实模式

在实模式下，寻址的最大范围是1M。

实模式下，只能访问地址在1M以下的内存称为常规内存，我们把地址在1M 以上的内存称为扩展内存。它是Intel公司80286及以后的x86(80386,80486和80586等)兼容处理器（CPU）的一种操作模式。实模式被特殊定义为20位地址内存可访问空间上，这就意味着它的容量是2的20次幂（1M）的可访问内存空间（物理内存和BIOS-ROM），软件可通过这些地址直接访问BIOS程序和外围硬件。

实模式下处理器没有硬件级的内存保护概念和多道任务的工作模式。但是为了向下兼容，所以80286及以后的x86系列兼容处理器仍然是开机启动时工作在实模式下。80186和早期的处理器仅有一种操作模式，就是后来我们所定义的实模式。实模式虽然能访问到1M的地址空间，但是由于BIOS的映射作用（即BIOS占用了部分空间地址资源），所以真正能使用的物理内存空间（内存条），也就是在640k到924k之间。1M地址空间组成是由16位的段地址和16位的段内偏移地址组成的。用公式表示为：物理地址=左移4位的段地址+偏移地址。

80286处理器体系结构引入了地址保护模式的概念，处理器能够对内存及一些其他外围设备做硬件级的保护设置（保护设置实质上就是屏蔽一些地址的访问）。使用这些新的特性，然而必不可少一些额外的在80186及以前处理器没有的操作规程。自从最初的x86微处理器规格以后，它对程序开发完全向下兼容，80286芯片被制作成启动时继承了以前版本芯片的特性，工作在实模式下，在这种模式下实际上是关闭了新的保护功能特性，因此能使以往的软件继续工作在新的芯片下。直到今天，甚至最新的x86处理器都是在计算机加电启动时都是工作在实模式下，它能运行为以前处理器芯片写的程序。

## 保护模式

保护模式本身是80286及以后兼容处理器序列之后产成的一种操作模式，它具有许多特性设计为提高系统的多道任务和系统的稳定性。例如内存的保护，分页机制和硬件虚拟存储的支持。现代多数的x86处理器操作系统都运行在保护模式下，包括Linux, Free BSD, 和Windows3.0（它也运行在实模式下，为了和Windows 2.x应用程序兼容）及以后的版本。

80286及以后的处理器另一种工作模式是实模式（仅当系统启动的一瞬间），本着向下兼容的原则屏蔽保护模式特性，从而容许老的软件能够运行在新的芯片上。作为一个设计规范，所有的x86系列处理器，除嵌入式Intel80387之外，都是系统启动工作在实模式下，确保遗留下的操作系统向下兼容。它们都必须被启动程序（操作系统程序最初运行代码）重新设置而相应进入保护模式的，在这之前任何的保护模式特性都是无效的。在现代计算机中，这种匹配进入保护模式是操作系统启动时最前沿的动作之一。

## 实模式与保护模式的区别

保护模式同实模式的根本区别是进程内存受保护与否。可寻址空间的区别只是这一原因的果。

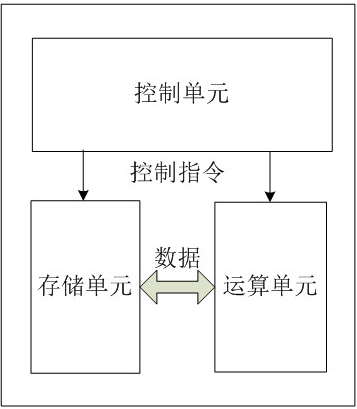
实模式将整个物理内存看成分段的区域,程序代码和数据位于不同区域，系统程序和用户程序没有区别对待，而且每一个指针都是指向真实的物理地址。这样一来，用户程序的一个指针如果指向了系统程序区域或其他用户程序区域，并改变了值，那么对于这个被修改的系统程序或用户程序，其后果就很可能是灾难性的。为了克服这种低劣的内存管理方式，处理器厂商开发出保护模式。这样，物理内存地址不能直接被程序访问，程序内部的地址（虚拟地址）要由操作系统转化为物理地址去访问，程序对此一无所知。至此，进程（这时我们可以称程序为进程了）有了严格的边界，任何其他进程根本没有办法访问不属于自己的物理内存区域，甚至在自己的虚拟地址范围内也不是可以任意访问的，因为有一些虚拟区域已经被放进一些公共系统运行库。这些区域也不能随便修改，若修改就会有: SIGSEGV（linux 段错误）;非法内存访问对话框（windows 对话框）。

在实模式下，内存段的处理是通过和段寄存器相关联的内部机制来处理的，这些段寄存器（CS、DS、 SS和ES）的内容形成了物理地址的一部分。具体来说，最终的物理地址是由16位的段地址和16位的段内偏移地址组成的。用公式表示为：物理地址=左移4位的段地址+偏移地址。

在保护模式下，段是通过一系列被称之为“描述符表”的表所定义的。段寄存器存储的是指向这些表的指针。用于定义内存段的表有两种：全局描述符表(GDT) 和局部描述符表(LDT)。GDT是一个段描述符数组，其中包含所有应用程序都可以使用的基本描述符。在实模式中，段长是固定的(为64KB)，而在保护模式中，段长是可变的，其最大可达4GB。LDT也是段描述符的一个数组。与GDT不同，LDT是一个段，其中存放的是局部的、不需要全局共享的段描述符。每一个操作系统都必须定义一个GDT，而每一个正在运行的任务都会有一个相应的LDT。每一个描述符的长度是8个字节。当段寄存器被加载的时候，段基地址就会从相应的表入口获得。描述符的内容会被存储在一个程序员不可见的影像寄存器(shadow register)之中，以便下一次同一个段可以使用该信息而不用每次都到表中提取。物理地址由16位或者32位的偏移加上影像寄存器中的基址组成。

# CPU的内部架构和工作原理

我们都知道CPU的根本任务就是执行指令，对计算机来说最终都是一串由“0”和“1”组成的序列。CPU从逻辑上可以划分成3个模块，分别是控制单元、运算单元和存储单元，这三部分由CPU内部总线连接起来。如下所示：

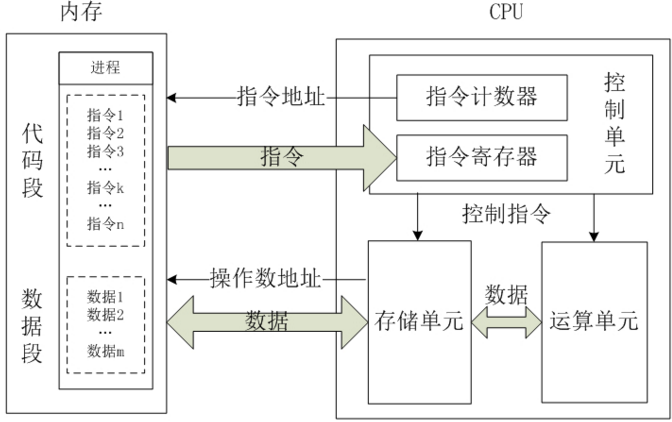


控制单元：控制单元是整个CPU的指挥控制中心，由指令寄存器IR(Instruction Register)、指令译码器ID(Instruction Decoder)和操作控制器OC(Operation Controller)等，对协调整个电脑有序工作极为重要。它根据用户预先编好的程序，依次从存储器中取出各条指令，放在指令寄存器IR中，通过指令译码(分析)确定应该进行什么操作，然后通过操作控制器OC，按确定的时序，向相应的部件发出微操作控制信号。操作控制器OC中主要包括节拍脉冲发生器、控制矩阵、时钟脉冲发生器、复位电路和启停电路等控制逻辑。

运算单元：是运算器的核心。可以执行算术运算(包括加减乘数等基本运算及其附加运算)和逻辑运算(包括移位、逻辑测试或两个值比较)。相对控制单元而言，运算器接受控制单元的命令而进行动作，即运算单元所进行的全部操作都是由控制单元发出的控制信号来指挥的，所以它是执行部件。

存储单元：包括CPU片内缓存和寄存器组，是CPU中暂时存放数据的地方，里面保存着那些等待处理的数据，或已经处理过的数据，CPU访问寄存器所用的时间要比访问内存的时间短。采用寄存器，可以减少CPU访问内存的次数，从而提高了CPU的工作速度。但因为受到芯片面积和集成度所限，寄存器组的容量不可能很大。寄存器组可分为专用寄存器和通用寄存器。专用寄存器的作用是固定的，分别寄存相应的数据。而通用寄存器用途广泛并可由程序员规定其用途，通用寄存器的数目因微处理器而异。这个是我们以后要介绍这个重点，这里先提一下。

我们将上图细化一下，可以得出CPU的工作原理概括如下：



总的来说，CPU从内存中一条一条地取出指令和相应的数据，按指令操作码的规定，对数据进行运算处理，直到程序执行完毕为止。

CPU的运行原理就是：控制单元在时序脉冲的作用下，将指令计数器里所指向的指令地址(这个地址是在内存里的)送到地址总线上去，然后CPU将这个地址里的指令读到指令寄存器进行译码。对于执行指令过程中所需要用到的数据，会将数据地址也送到地址总线，然后CPU把数据读到CPU的内部存储单元(就是内部寄存器)暂存起来，最后命令运算单元对数据进行处理加工。周而复始，一直这样执行下去，天荒地老，海枯枝烂，直到停电。

# BIOS

BIOS（Basic Input Output System），是一组固化到计算机内主板上一个ROM芯片上的程序，它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序，它可从CMOS中读写系统设置的具体信息。 其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。一般它是一块32针的双列直插式的集成电路，上面印有"BIOS"字样。

BIOS芯片中主要存放：

● 自诊断程序：通过读取CMOSRAM中的内容识别硬件配置，并对其进行自检和初始化；

● CMOS设置程序：引导过程中，用特殊热键启动，进行设置后，存入CMOS RAM中；

● 系统自举装载程序：在自检成功后将磁盘相对0道0扇区上的引导程序装入内存，让其运行以装入操作系统；

● 主要I/O设备的驱动程序和中断服务：由于BIOS直接和系统硬件资源打交道，因此总是针对某一类型的硬件系统，而各种硬件系统又各有不同，所以存在各种不同种类的BIOS，随着硬件技术的发展，同一种BIOS也先后出现了不同的版本，新版本的BIOS比起老版本来说，功能更强。

# 中断向量表

计算机内存的前1024个字节（偏移量00000H到003FFH）保存着256个中断向量，每个中断向量占4个字节，前两个字节保存着中断服务程序的入口地址偏移量（段内偏移量），后两个字节保存着中断程序的入口段地址（段基址），使用时，只要将它们分别调入寄存器IP及CS中，就可以转入中断服务程序实现中断调用。每当中断发生时，CPU将中断号（硬件产生的中断标识码）乘以4，在中断向量表中得到该中断向量地址，进而获得IP及CS值，从而转到中断服务程序的入口地址，调用中断服务程序。

80x86系统是把所有的中断向量集中起来，按中断类型号从小到大的顺序存放到存储器的某一区域内，这个存放中断向量的存储区叫做中断向量表，即中断服务程序入口地址表。

由于中断向量表可以在操作系统层面灵活修改，因此，不同的系统的中断向量表可能是不同的。此外，intel在CPU的保护模式下，占用了0x00 ~ 0x1F共32个中断号，在Linux下，是从0x20开始用于系统自身的中断的，包括8259芯片的中断重置。

# 操作系统的启动过程

1. 接通电源；
2. CPU的硬件逻辑电路加电后最先把CS的值设为0xFFFF，把IP的值设为0x0000，这样CS:IP就指向了0Xffff0这个位置，而0Xffff0就是BIOS的首地址。CS:IP中存放的值就是将要执行的指令的地址。所以这一步，BIOS被启动运行；

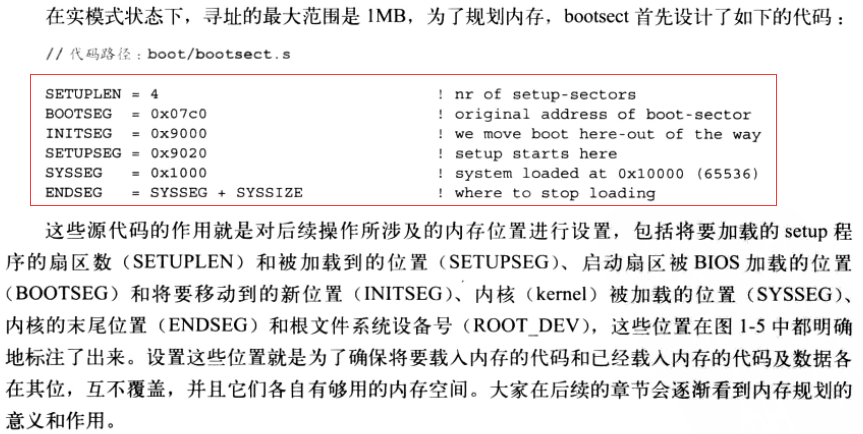
这是纯硬件的动作，所有操作都是固定在硬件中的，如果这一步执行失败，系统直接死机。

CS：代码段寄存器，记录代码段的起始位置；IP：指令指针寄存器，记录指令在段内的偏移地址

1. BIOS被启动。BIOS被固化在主板上的一块很小的ROM芯片中。

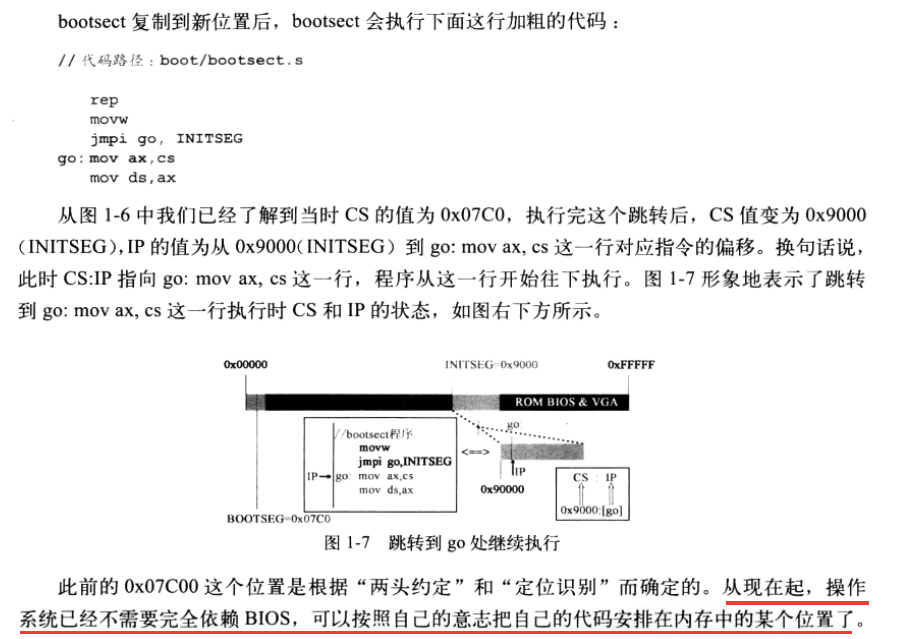
* BIOS检测显卡、内存等；
* BIOS在内存的开始位置（0x00000）用1KB的空间创建中断向量表。中断向量表中有256个中断向量，每个中断向量占4个字节，其中两个字节是CS的值，两个字节是IP的值，每个中断向量都指向一个具体的中断服务程序；
* 接着用256B(0x00400-0x004FF)的空间创建BIOS数据区，
* 然后在56KB之后的位置加载了8KB的中断服务程序

1. BIOS加载引导扇区。CPU接收到int 0x19中断，执行其指向的中断服务程序，把磁盘0磁道1扇区的引导程序拷贝到内存0x07C00处。这个第一扇区就是启动扇区。这是计算机开机以来内存中第一次有了操作系统的代码，从此计算机开始与硬盘上的操作系统产生联系。启动扇区的程序由bootsect.s中的汇编程序汇编而成。
2. bootsect.s规划内存

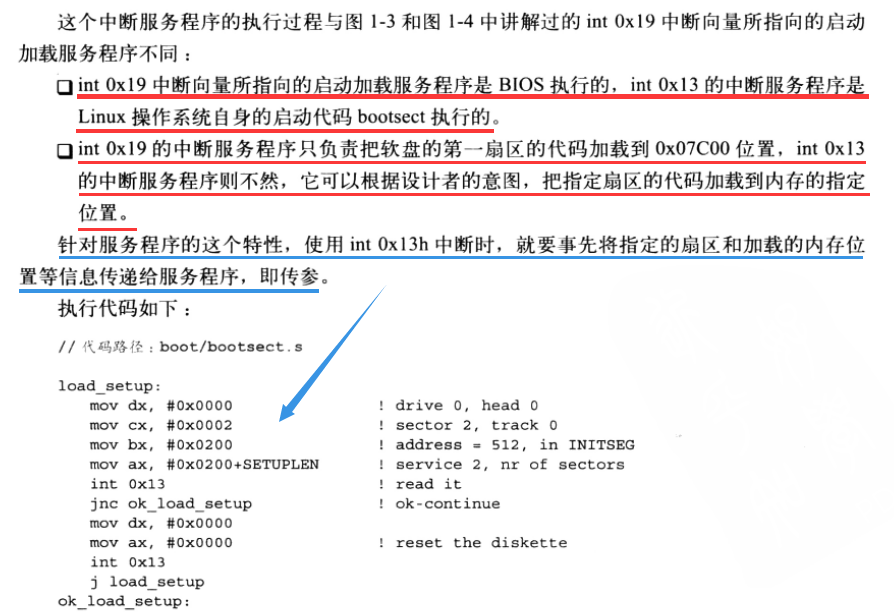


1. 复制bootsect.s。bootsect.s把自身的全部内容复制到0x90000处，然后进行跳转，





1. 加载setup程序。借助BIOS int 0x13中断向量指向的中断服务程序，bootsect把setup.s（位于第二扇区开始的4个扇区）程序载入内存。



1. 加载system模块。借助BIOS int 0x13中断向量指向的中断服务程序，bootsect使用read\_it子程序把system模块的代码（第6个扇区开始的共240个扇区大小）载入内存。
2. bootsect检测根设备号。至此，bootsect的工作全部完成。后续由setup程序接着bootsect运行。
3. setup运行。至此，操作系统内核的加载工作完成。系统从实模式转为保护模式。

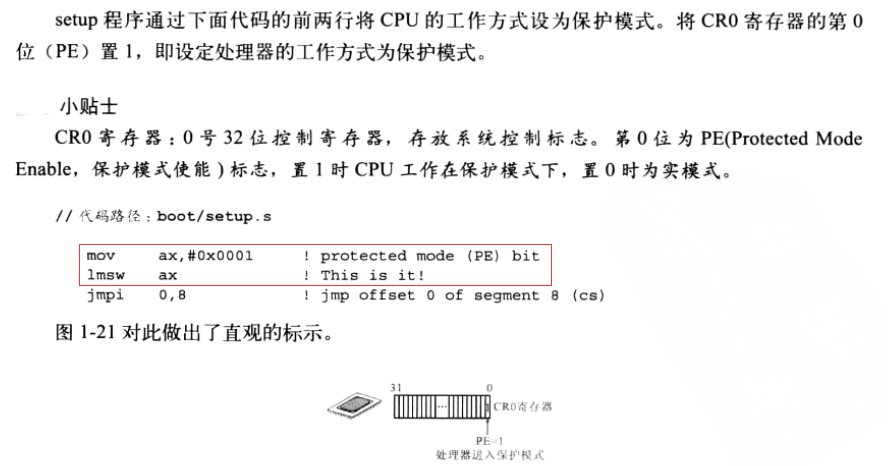
* 从设备上提取内核运行所需的机器系统数据

1. 关闭BIOS中断，并将系统内核代码拷贝至0x00000处，覆盖掉BIOS原来所在的数据区域和以及BIOS创建的中断向量表区域，让操作系统的内核代码位于内存的最开始位置。直到新的系统中断服务体系建立之前，系统不再响应任何中断。
2. setup对中断描述符表寄存器（IDTR）和全局描述表寄存器（GDTR）进行初始化。

由于此时内核尚未真正运行起来，还没有进程，所以GDT的第1项为空，第2项为内核代码描述符，第3项为数据段描述符，其余项都为空。

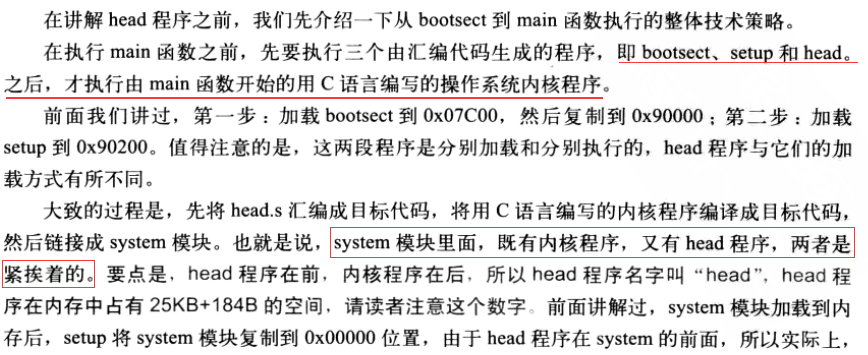
* GDT是唯一存放段寄存器内容的数组，配合系统在保护模式下的寻址操作，其存放着每一个任务局部描述符表（LDT）地址和任务状态段地址，用于完成进程中各段的寻址、现场保护和现场回复。
* GDTR存放GDT的入口地址。
* IDT存放保护模式下所有中断服务程序的入口地址。IDT此时为空表。
* IDTR存放IDT的入口地址。

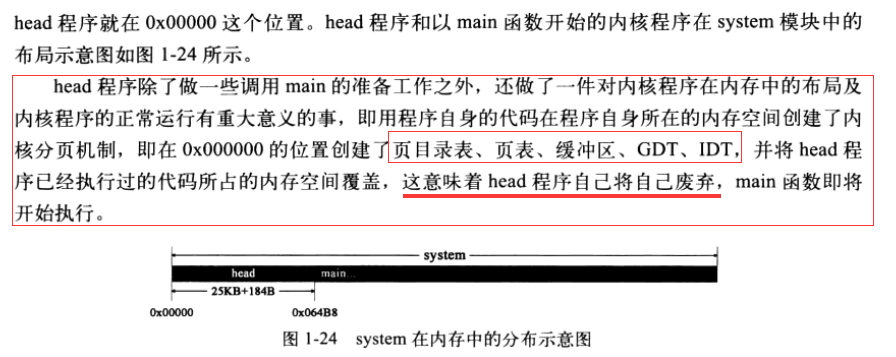
1. 打开A20实现32位寻址。
2. setup将CPU设为保护模式



至此，setup程序执行完毕。

1. head.s开始执行。





* 设置中断描述符表GDT和GDTR的值
* 创建页表目录和页表
* 内核的分页机制构建完成
* head程序的最后一步：ret，跳入main函数开始执行 ps：此处仍处于关闭中断的状态

1. 操作系统的main函数开始运行，开机过程完成。

# 从main到怠速

怠速是指：操作系统已经完成了所有的工作，随时可以响应用户的操作。

系统达到怠速状态前所做的一切准备工作的核心目的就是让用户程序能够以“进程”的方式正常运行。

1. 创建进程0，并让进程0具备在32位模式下在主机中运算的能力。

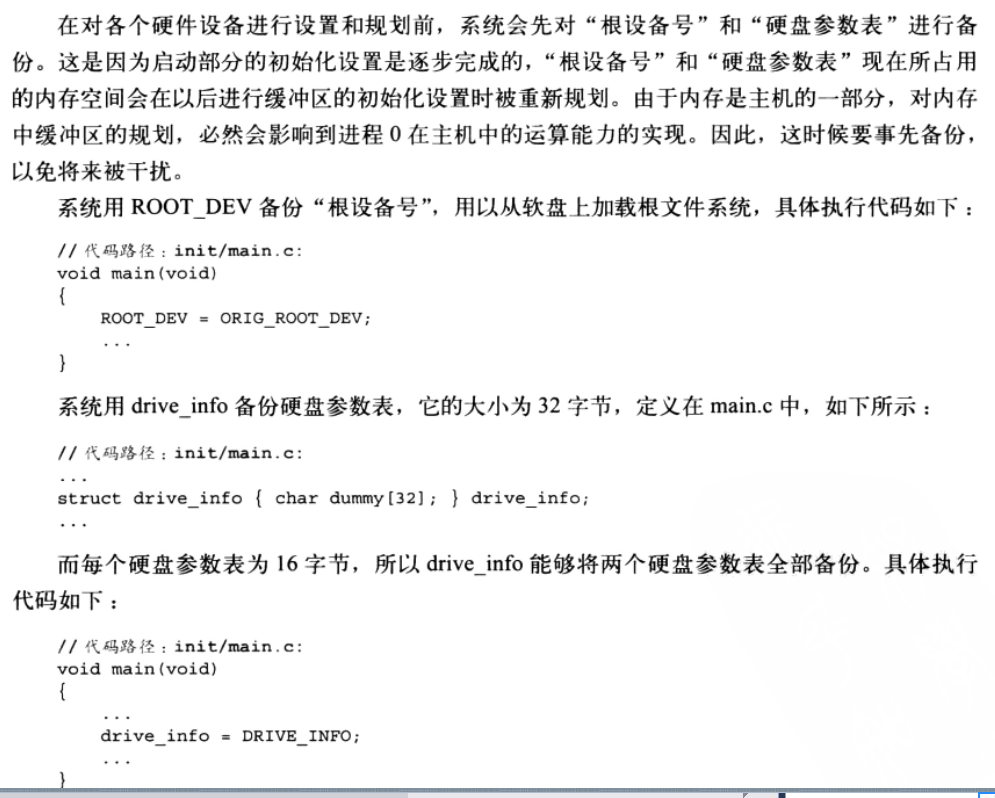
每一个进程都是由一个已经存在的进程创建的，并且拥有父进程的所有功能，所以必须先创建一个最原始的进程，即进程0。

1. 用进程0创建进程1，使进程1不仅仅具备进程0的功能，并且还能以文件的形式与外设进行数据交互。
2. 用进程1创建进程2，使进程2不仅仅具备进程1的功能，并且还进一步具备支持“人机交互”的能力，最终实现怠速。

进程2负责构建一套人机交互界面，以支持用户与计算机的交互，最终进入怠速状态，由此，进程2就成为创建第一个用户进程的母体，它所具备的能力将为用户程序以“进程”的形式正常运行提供保障。

# 开中断之前的准备工作

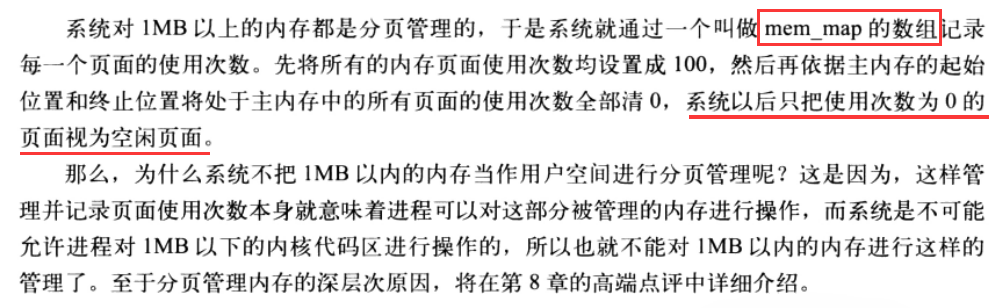
## 11.1 复制根设备号和硬盘参数表



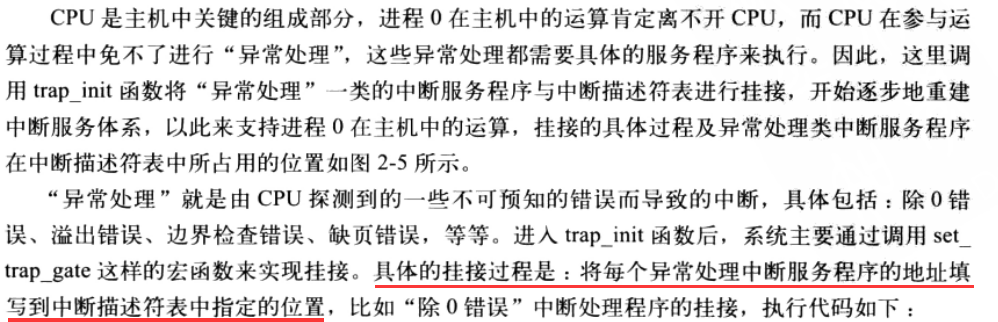
## 11.2 物理内存格局规划

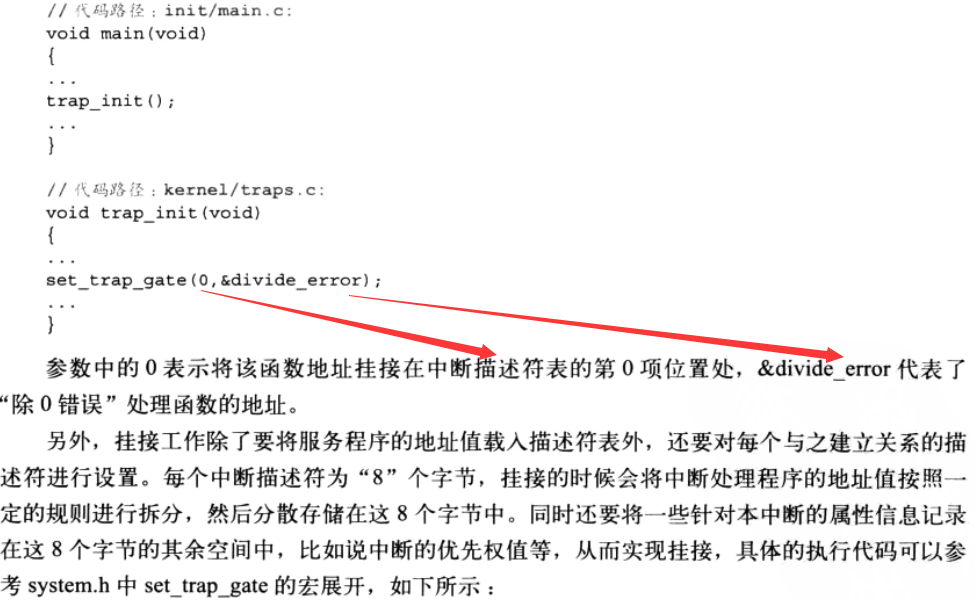


## 11.3 初始化内存管理结构mem\_map



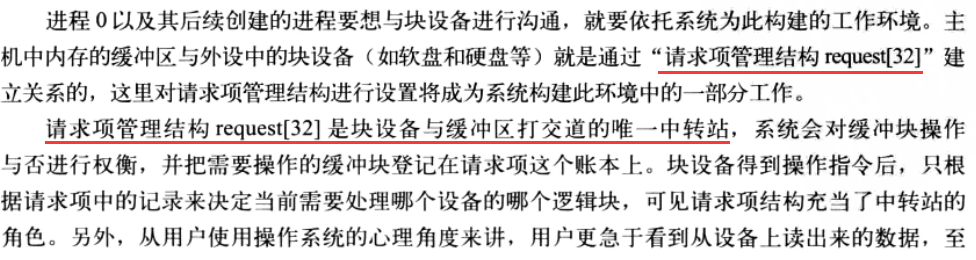
## 11.4 挂接异常处理类中断服务程序

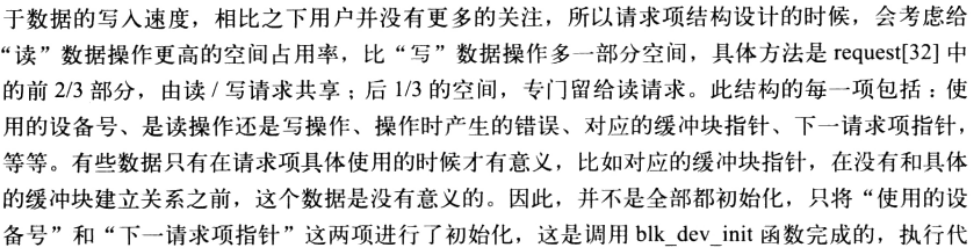




32位保护模式下的中断服务体系就是通过不断建立这种挂接关系形成的，这种32位中断服务体系是为适应一种被动响应中断信号的机制而建立的。

## 11.5 初始化块设备请求项结构

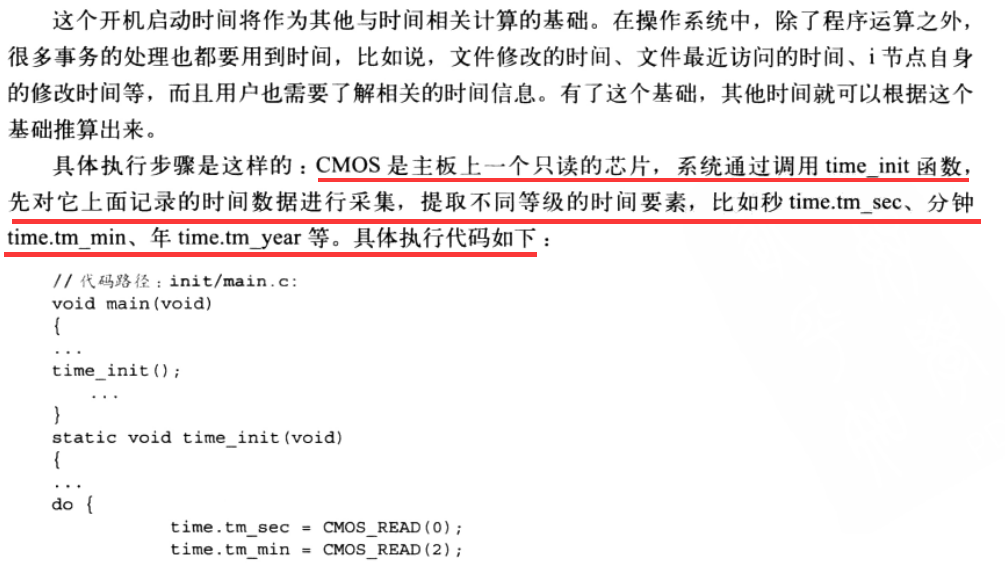


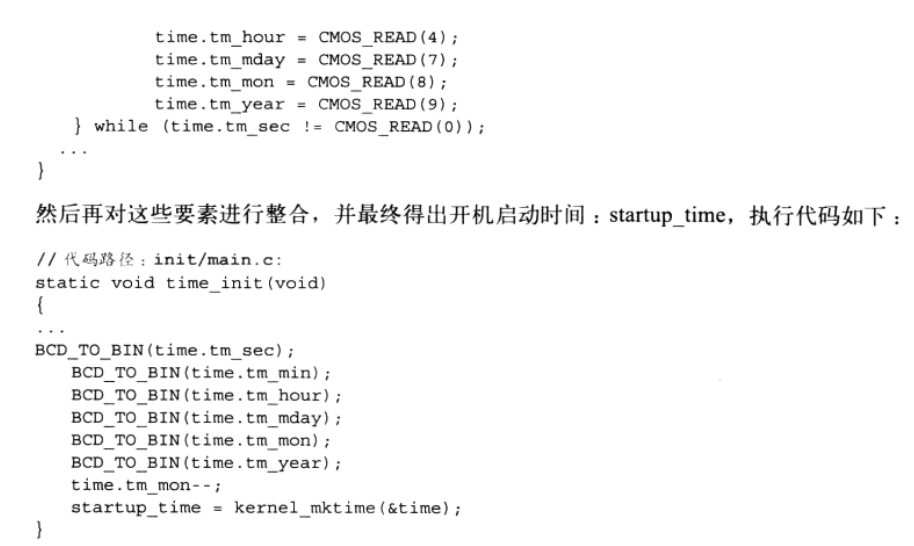


## 11.6 挂接与人机交互界面相关的外设的中断服务程序

例如 键盘、鼠标、显示器等。

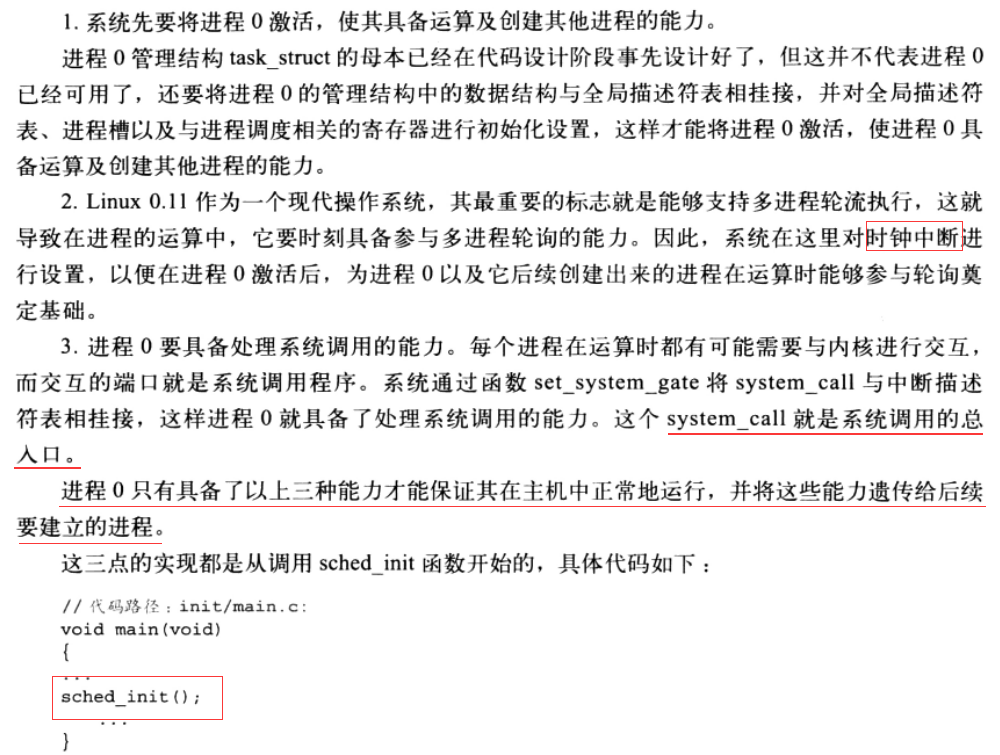
## 11.7 开机启动时间设置





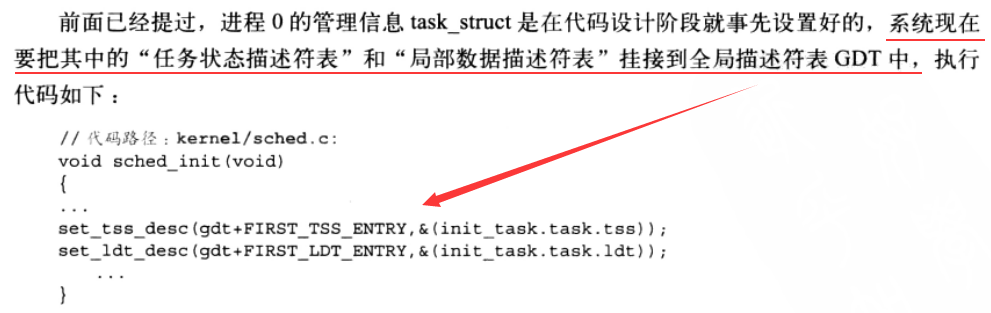
## 11.8 激活进程0

进程0的数据结构代码已经在代码设计阶段事先设计好了，此时要做的是把进程0的数据结构与全局描述表挂载，并对相关的寄存器进行设置。

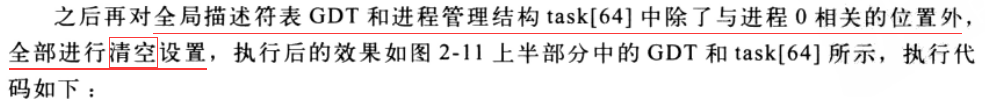


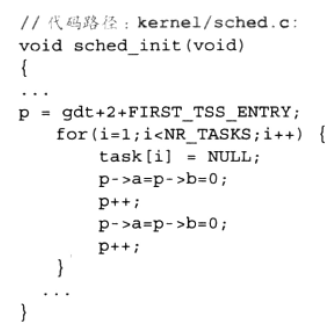
## 11.9 初始化进程相关事务设置

1. 挂载

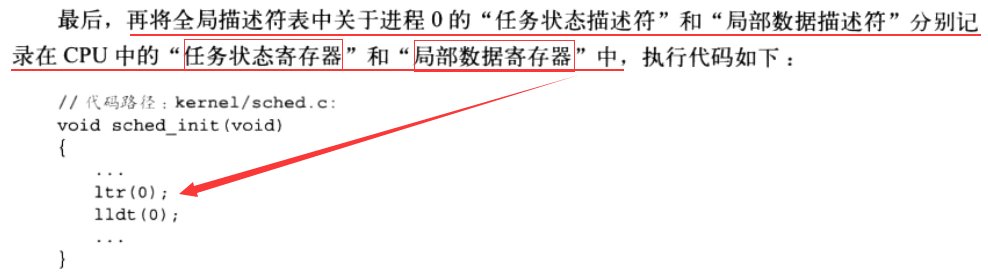


1. 清零





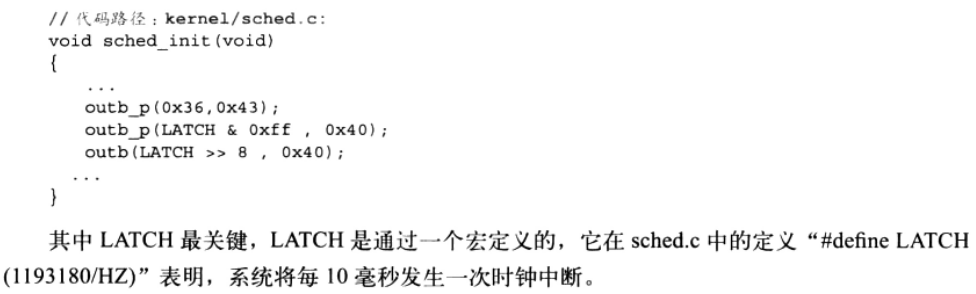
1. 设置寄存器



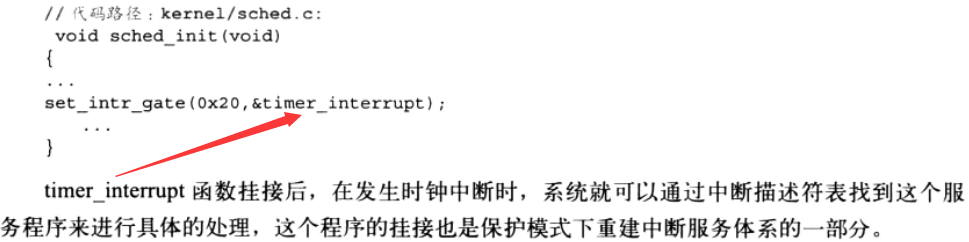
## 11.10 设置时钟中断

时钟中断是进程0和其他由其创建的进程的轮询基础。

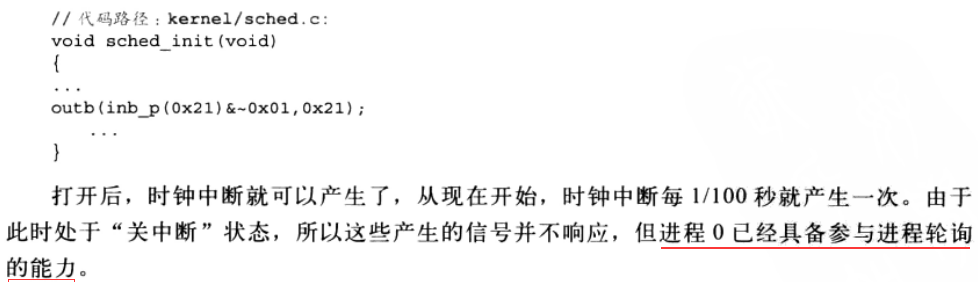
* 设置支持轮询的定时器



* 设置与轮询相关的服务进程



* 打开芯片中与时钟中断相关的屏蔽码



## 11.11 挂载系统调用服务程序

将系统调用处理函数set\_system\_gate与中断描述符表相挂载。system\_call是整个操作系统中系统调用软中断的总入口，用户程序产生的所有系统调用都是通过这个总入口找到具体的系统调用函数。

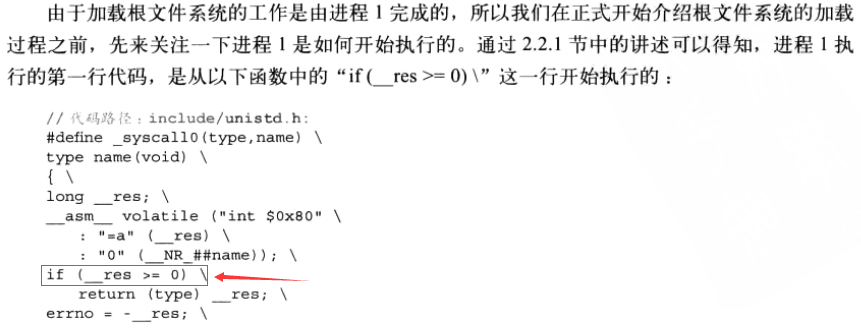
|  |
| --- |
| //代码路径：kernel/sched.c  void sched\_init(void)  {  set\_system  } |

# 进程1执行过程

进程1是由进程0创建并交换给执行权的。

在此之前，进程0已经构建了与硬盘软盘的交互环境。

## 进程1开始执行

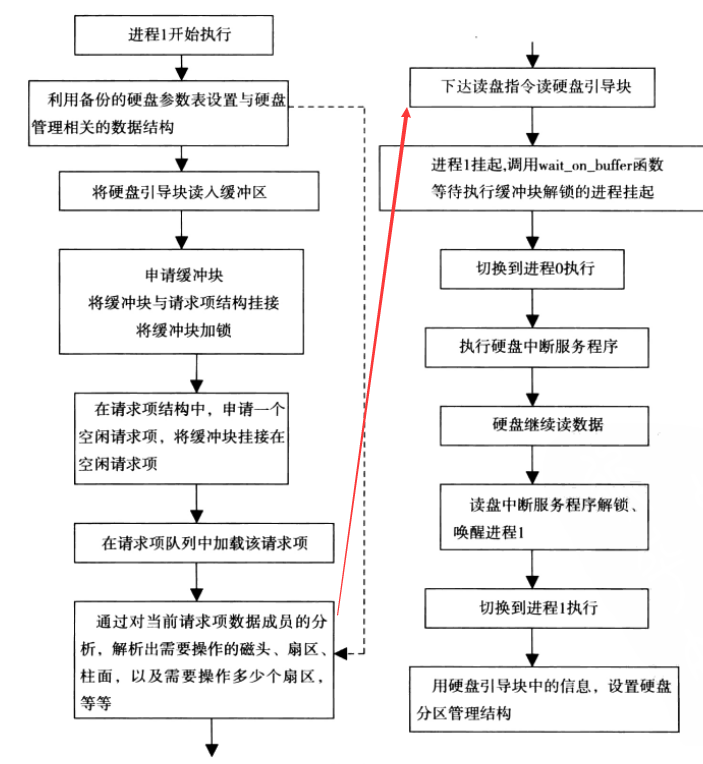


## 加载根文件系统

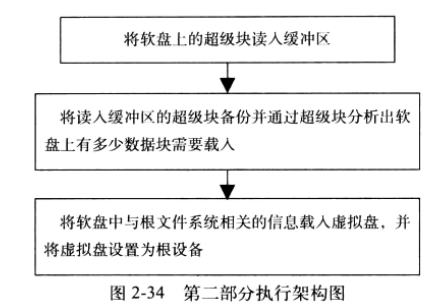
系统与外设以文件的方式进行交互从根本上是由根文件系统来支持的，所以必须先加载根文件系统。

### 设置与硬盘管理相关的数据结构

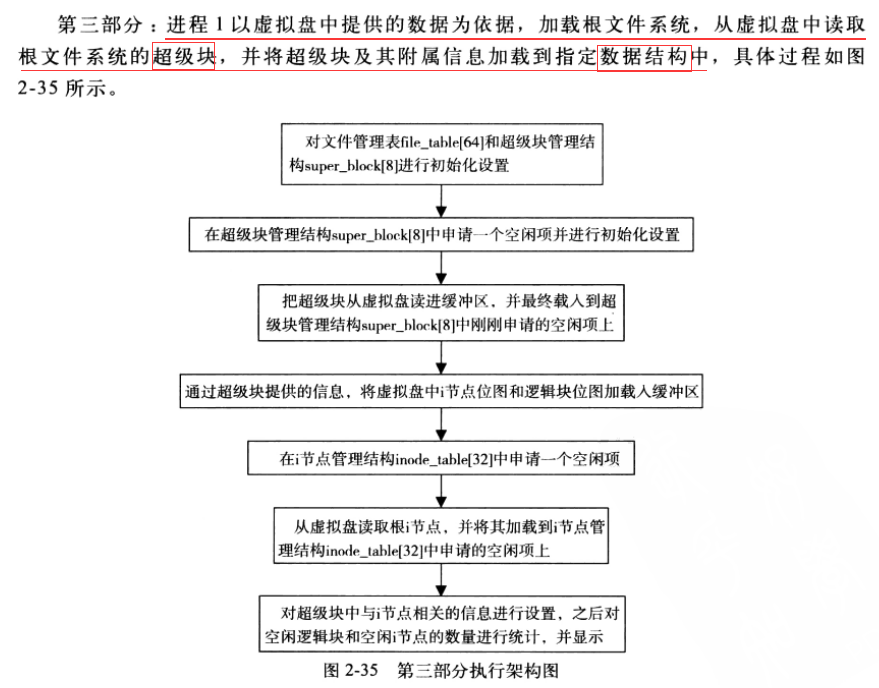
为系统与硬盘以文件形式交互奠定基础。



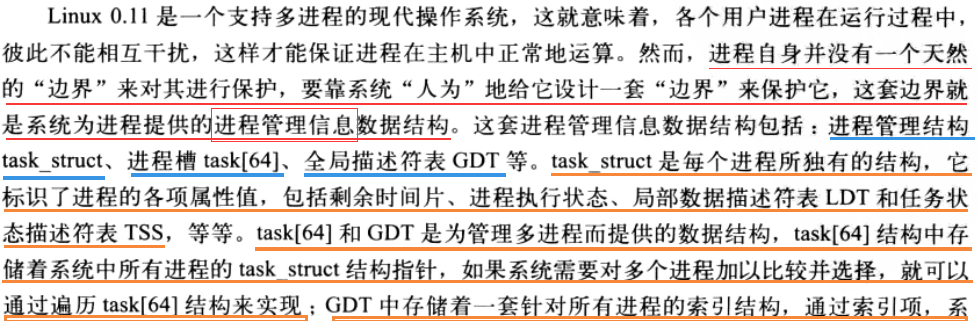
### 12.2.2 用虚拟磁盘代替软盘，使其成为根设备



### 加载根文件系统



# 进程





## 进程id

内核中使用全局变量last\_pid来系统自开机以来存放的进程数量，同时用last\_pid作为最新创建的进程的进程id

## 进程槽

内核中的全局变量task[64]

## 进程切换

* 时间片用完
* 逻辑执行需要切换

# Android移植

Android移植包括应用移植和系统移植。

使一个操作系统在特定硬件平台上运行的一个首要条件就是该操作系统支持该硬件平台的CPU架构。

Android移植在很大程度上是Linux内核的移植，Linux内核的移植主要是移植驱动程序，不同Linux版本的驱动程序不能通用，需要重新修改源代码并在新的Linux内核下重新编译才可以运行在新的Linux内核版本上。

Linux驱动和普通的Linux API没有本质区别，只是使用方式不同。

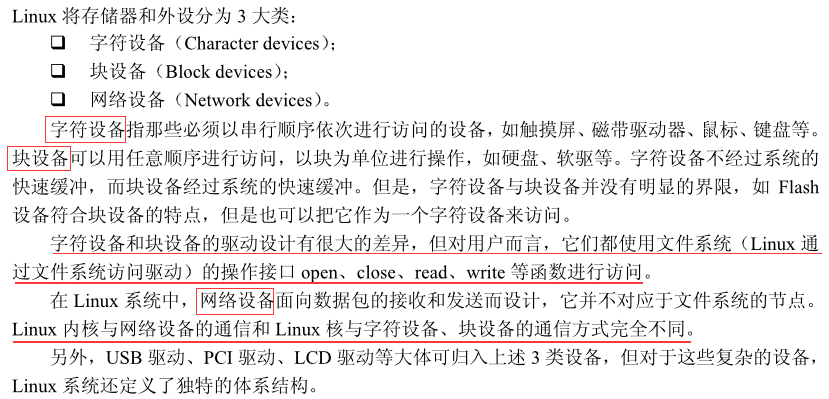
# 设备驱动

## 15.1 Linux设备驱动

应用程序在早期可以直接操作硬件，但是由于硬件类型种类繁多，一旦外部硬件发生变化，则应用程序就必须修改源代码重新编译。为了解决这个问题，驱动程序出现，驱动程序就是把所有的外部硬件进行抽象描述，封装成统一的访问接口，供上层的应用程序调用。

Linux驱动只与Linux内核有关，与Linux发行版的版本没关系，只要Linux内核一样，Linux驱动就可以在这些系统上通用。

Linux设备的分类：



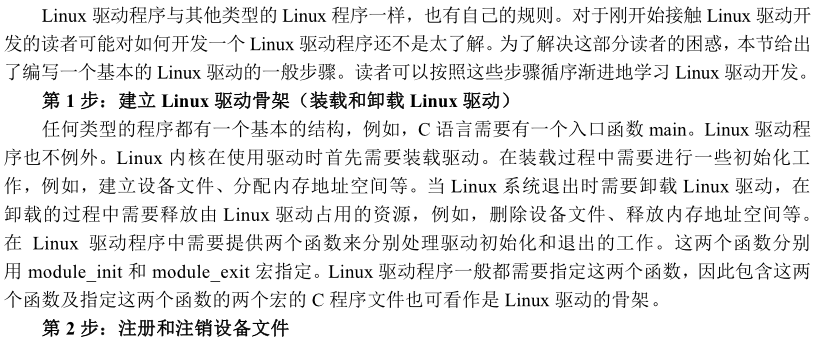
## 15.2 Linux内核版本号定义标准

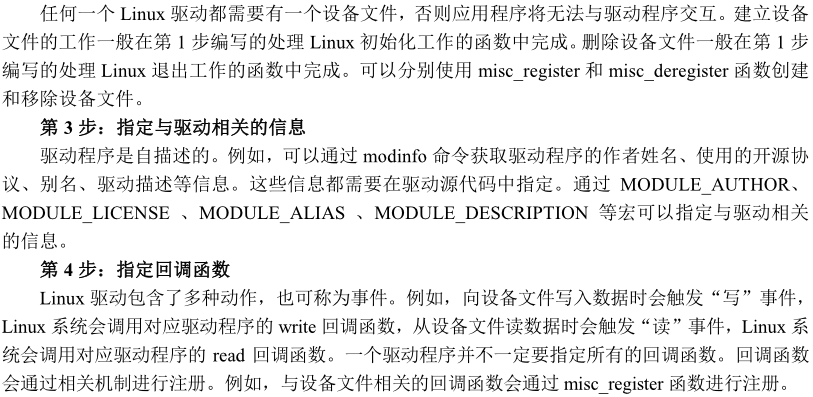


## 15.3 Linux驱动的工作方式和访问方式

* Linux驱动的工作方式和访问方式是Linux的亮点之一，受到了业界的广泛好评。
* Linux系统将每一个驱动都映射成一个文件，这些文件称为设备文件或驱动文件，存放在/dev目录中。这种设计理念使得与Linux驱动的交互就像与普通文件交互一样容易。由于大多数Linux驱动都有与其对应的设备文件，因此与Linux驱动交换数据就变成了与设备文件交换数据。例如：向Linux打印机驱动发送一个打印命令，就可以直接用C语言的open函数打开设备文件向里面写入数据。

## 15.4 Linux驱动程序开发的一般步骤







# 计算机运行模式

现代计算机都有两种运行模式：内核态（亦称 管态/核心态）和用户态。操作系统就是运行在内核态，在内核态模式下，操作系统具有对所有硬件的完全访问权，可以执行硬件能够运行的任何指令。计算机的这两种模式由程序状态字寄存器（PSW）中的一个二进制位控制。

在用户太模式下，有关I/O和内存保护的所有指令是被禁止使用的，将PSW中的模式位也是被禁止的。所以，为了使用系统服务，用户程序必须使用系统调用，系统调用陷入内核并调用操作系统功能，TRAP指令把用户态切换为内核态，系统调用完成后，系统调用后面的指令把控制权返回给用户程序并切换回用户态。

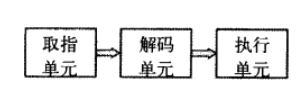
# 计算机硬件

计算机中的各部分硬件（CPU、内存等）都是通过总线与其他设备进行通信。

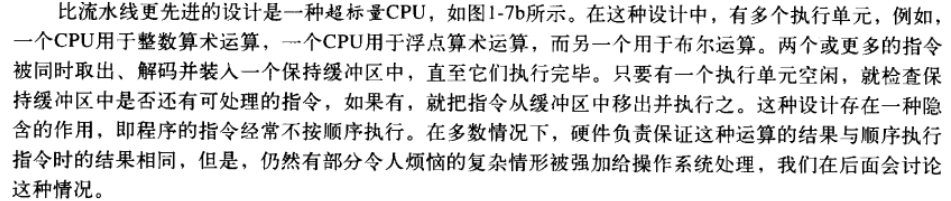
OS（操作系统）每次停止一个正在运行的程序时，必须保存所有的寄存器数据，以便在再次运行这个程序时，可以把这些寄存器数据重新载入执行。

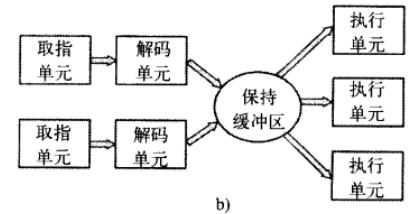
## 17.1 CPU

* 每个类型的CPU都有一套自己可执行的专门的指令集，在CPU的工作周期中就是循环从内存中读取指令并执行。
* 现代CPU都具有同时取出多条指令的机制，有分开的取指单元、解码单元和执行单元，同时可以对多条指令进行不同的分析操作，例如 当执行指令n时，可以对n+1指令解码并读取n+2指令，这种机制成为流水线机制。这种流水线设计机制使得指令一旦进入流水线就必须被执行完毕，及时其前面的指令是跳转指令，本条指令也必须执行完毕后才能跳转，这种机制增加了编译器和操作系统的复杂性：



* 为了解决流水线CPU的弊端，设计了更先进的超标量CPU：





## 17.2 寄存器

寄存器和CPU使用相同的材料制成，所以和CPU一样快，CPU访问寄存器是基本没有时间延迟的。

* 通用寄存器：因为CPU从内存获取指令的时间大于执行指令的时间，所以CPU内部都有一些用来保存关键变量和临时数据的寄存器。
* 程序计数器：保存下一条将要取出的指令的地址。
* 堆栈指针：指向内存中当前栈的顶端。当前栈含有已经进入但是还没有退出的每个过程的一个框架，该框架保存了有关的输入参数、局部变量以及没有保存在寄存器中的临时变量。
* 程序状态字：包含了条件码位（比较指令设置）、CPU优先级、模式（用户态&内核态）、以及各种其他控制。

## 17.3 磁盘

磁盘由于是一种机械装置，所以其速度较内存很慢，每个盘面都有一个读写头。



每块磁盘都有一个或者多个金属盘片，每个盘片分为两个盘面。在给定臂的位置，每个磁头可以读取一段环形区域，成为磁道。把一个给定臂的位置上的所有磁道合并起来，就组成一个柱面。每个磁道划分为若干个扇区（就是一段弧形区域，一般为512字节）。机械臂从一个柱面移动到相邻的柱面上需要1ms的时间，移动到随机的一个柱面上需要5ms-10ms的时间，一旦磁臂到达正确的磁道上，还需要等待所需的扇区旋转到磁头之下（这又增加了5ms-10ms的延时），才开读写数据。

## 17.4 I/O设备

I/O设备一般由两个部分：设备控制器和设备本身。设备控制器就是插在电路板上的一块芯片或一组芯片，这块电路板物理地控制着设备。

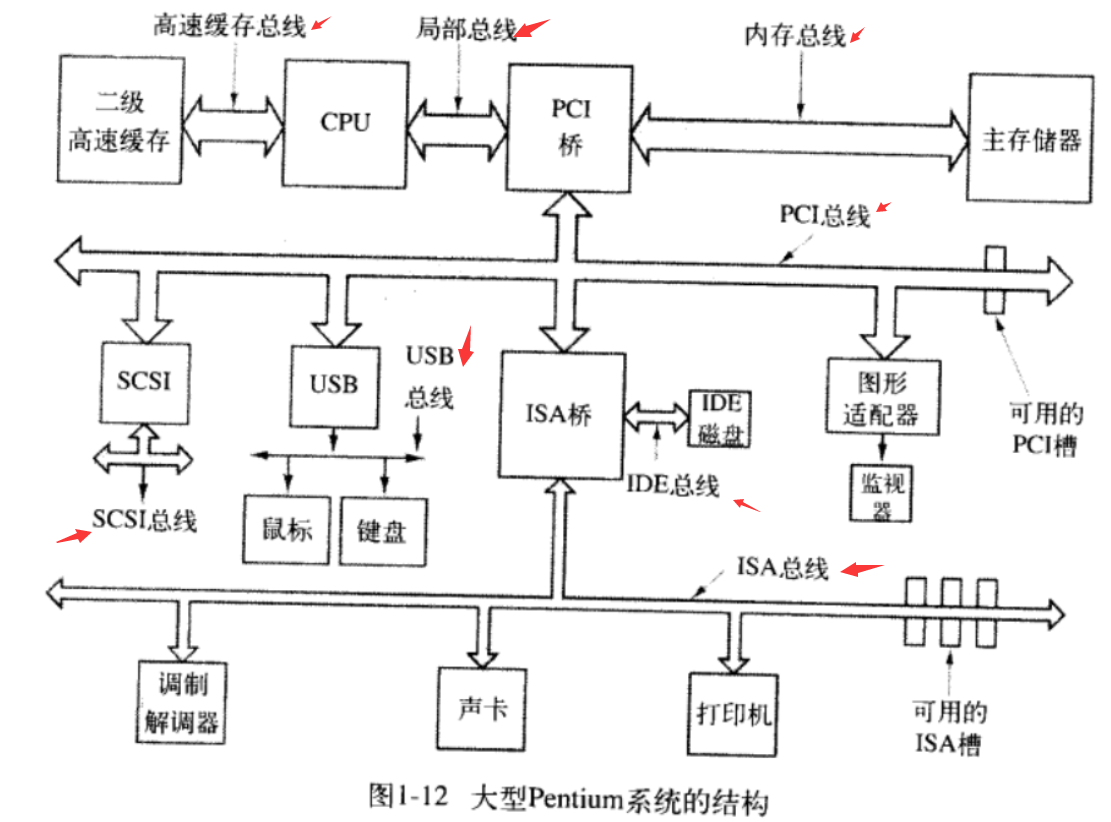
每类设备控制器都是不同的，所以需要不同的软件进行控制，专门与设备控制器进行交互，发送和接收命令，这种程序成为设备驱动程序。设备驱动程序要想使用就必须装入操作系统中，这样它就可以在内核态运行。

要想把设备驱动程序装入操作系统有三种方式：

* + - * 1. 将内核与设备驱动程序重新链接，然后重启系统。许多Unix系统以这种方式工作
        2. 在操作系统文件中设置一个入口，并通知该文件需要一个设备驱动程序，然后重启系统，系统重启时会自动去寻找所需的设备驱动程序并装载。Windows以这种方式工作
        3. 操作系统在运行时能够接收新的设备驱动程序并且立即将其安装好运行，无需重启系统。USB的热插拔技术

每个设备控制器都有少量的用于通信的寄存器，被称为I/O端口。

## 17.5 总线



### USB总线

即通用串行总线，是位串行总线，是一种集中式控制总线。用来连接所有慢速的I/O设备，诸如键盘、鼠标等。USB是一种集中式总线，其根设备每隔1ms轮询一次I/O设备，查看是否有信息收发。所有的USB设备共享一个USB设备驱动器，于是就不需要为新的USB设备安装新的设备驱动器了，所以无需重启计算机就可以添加新的USB设备。

### SCSI总线

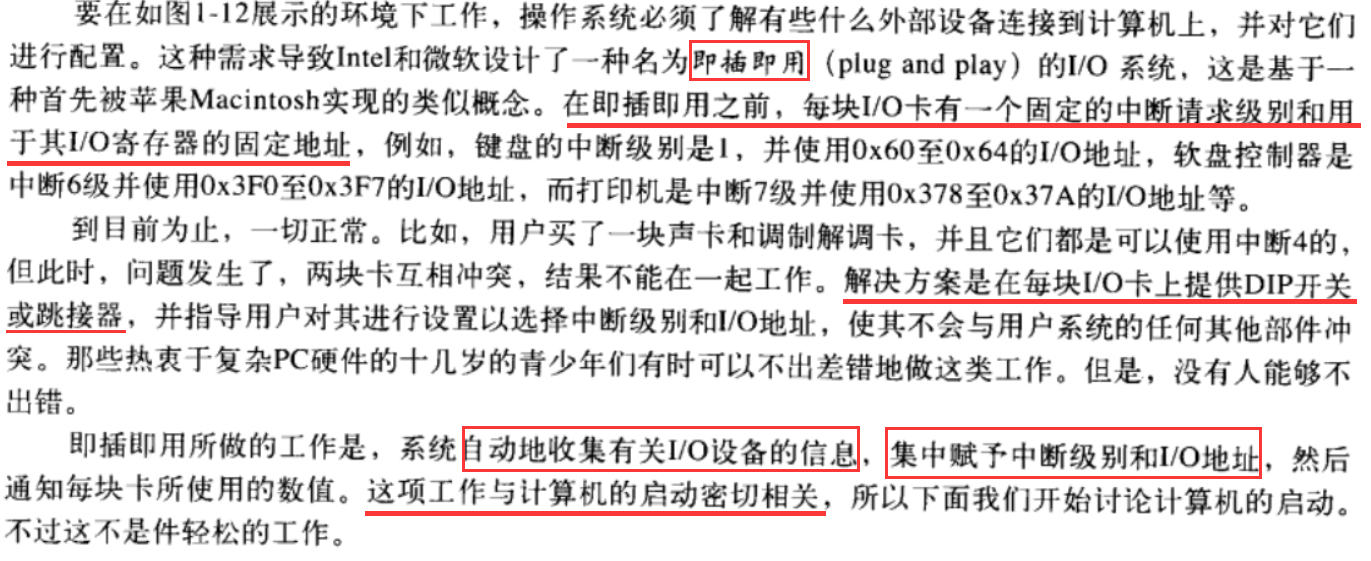
small computer system interface。这是一种高速总线，用在高速硬盘、扫描仪和其他需要较大宽带的设备上。

### IEEE总线

又称火线，是位串行总线。不需要集中式控制器。适合将数码相机和类似的多媒体设备连接到计算机上。

## 17.6 即插即用设计原理

在正常的工作情况下，操作系统必须知道有哪些外部设备连接到了计算机上，并对他们进行配置。



# 操作系统

## 进程

进程本质上是正在执行的一个程序，是容纳运行一个程序所需的所有信息的容器，进程是操作系统进行资源分配的基本单位。

与进程最关键的系统调用是创建进程和终止进程的系统调用。

在Unix系统中，fork系统调用是创建进程的唯一途径，它创建一个父进程的精确副本，包含所有的文件描述符、寄存器等，所有的变量都具有相同的值，fork之后父进程和子进程就分开了，以后其中一个的变化不会影响到另一个（父/子进程共享程序正文 ps:程序正文即程序代码段）。

### 18.1.1 进程表

进程表是操作系统的一张表，是数组或链表结构，表中的每一项就是一个进程的记录。

## 18.2 线程

线程是进程内部的不同执行路径，是cpu调度的基本单位。

* 线程睡眠：即线程停止执行，让出cpu的执行权；

## 18.3文件系统

操作系统的一项主要功能就是隐藏磁盘和其他I/O设备的细节特性，并提供给程序员一个良好、清晰的独立于设备的抽象文件模型。

在读写文件之前，首先要打开文件，检查文件的访问权限，若权限允许，系统返回一个小整数，称之为文件描述符。若禁止访问，系统返回一个错误码。

在Unix系统中，每一个文件都有一个唯一的编号用户标识文件。

### 18.3.1 特殊文件

在Unix系统中的特殊文件是为了使I/O设备看起来像文件一般，这样，I/O设备就可以像读普通文件那样使用系统调用进行读写。

块特殊文件：指可随机存储的块组成的设备，如磁盘。程序可以直接访问设备的每一块而

不必考虑存放该文件的文件系统结构；

字符特殊文件：指接收和输出字符流的设备，如打印机、调制解调器等。

## 18.4 管道

管道是一种虚文件，它可以连接两个进程使其之间相互通信。

## 18.5 文件/信息安全保护

Unix系统对每个文件赋予一个9位的二进制保护码实现对文件的保护。保护码分为

组：第1组用于所有者，第2组用于同组的其他用户，第3组用于其他组成员，每一部分的权限都是3个二进制位r（读）、w（写）、x（执行），“-”表示禁止对应的权限。

## 18.6 系统调用

引发系统调用的实际机制是非常依赖于机器的，并且必须用汇编代码来表达。用户执行系统调用时会陷入内核，把控制权转移给操作系统，操作系统检查参数并找出所需的调用进程然后执行，执行完毕后返回结果并把控制权转交给用户程序。

系统调用过程：1）调用程序首先把参数逆序压入堆栈（即从右到左读取参数，这是由

C/C++编译器的历史原因决定的），即最后一个参数位于栈顶；

2）调用库函数。库函数一般由汇编代码写成，系统调用编号放在寄存

器中；

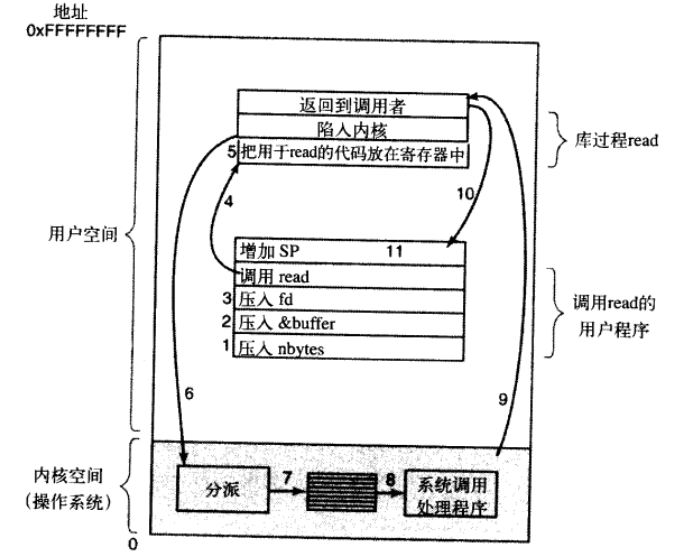
3）执行TRAP指令，把用户态切换到内核态，在内核中的一个固定地址开始

执行；

4）跟随在TRAP指令后面的内核代码开始检查系统调用编号，系统调用编

号指向一个系统调用处理器的指针；

5）系统调用句柄运行，运行完毕后返回到用户空间。



## 18.7 Windows Win32 API

## 18.8 内存

### 18.8.1 虚拟内存

虚拟内存通过在RAM和磁盘之间反复移动信息块的方式，提供了运行比物理内存大的程序的能力；

虚拟内存还使得程序在运行时动态链接库，而不是在编译时链接；

## 18.9