我的读书笔记

zhaoshulin.cn@gmail.com

May 13, 2014

Contents

1 《Linux内核设计的艺术》

1.1 第一章：从开机加电到执行main函数之前的过程

1.1.1 开机之后，从启动盘加载OS的三个步骤是？

(1) 启动BIOS，准备实模式下的中断向量表和中断服务程序；

(2) 利用上一步的中断服务程序，从启动盘加载OS到内存；

(3) 为执行32位的main函数做过渡工作。

1.1.2 什么是实模式（Real Mode）？

实模式的特性是一个20位的存储器地址空间（$2^{20} = 1048576$，即1MB的存储器可被寻址），可以直接软件访问BIOS以及周边硬件，没有硬件支持的分页机制和实时多任务概念。

1.1.3 BIOS的启动原理？

从硬件角度看，Intel 80x86 系列的CPU可以分别在16位实模式和32位保护模式下运行。为了解决最开始的启动问题，x86的cpu的硬件都设计为加电即进入16位实模式状态运行。在加电瞬间，强行将CS 置为0xF000，IP置为0xFFF0，这样，CS:IP=0xFFFF0，而BIOS程序的入口地址恰恰就是0xFFFF0。

1.1.4 BIOS最早对内存做了什么手脚？

(1) 中断向量表

(2) BIOS数据区

(3) 中断服务程序

1.1.5 如何加载第一部分内核代码（boot sect）？

完成自检之后，体系结构与BIOS联手，会让CPU接收到一个int 0x19中断，该中断的服务程序把0磁头0磁道1扇区中的bootsect.s复制到内存0x07C00处。

1.1.6 如何加载第二部分内核代码（setup）？

(1) 规划内存，设置SETUPLEN, SETUPSEG, BOOTSEG, INITSEG, INITSEG, SYSSEG, ENDSEG, ROOT_DEV的内存位置；

(2) bootsect.s将自身从BOOTSEG（0x07c00）处复制到INITSEG(0x9000)处；

(3) 执行0x int 13中断，将setup.s加载到内存的SETUPSEG处。

1.1.7 如何加载第三部分内核代码（system模块）？

bootsect调用read_it 子程序，将system模块加载到内存的SYSSEG处。

1.1.8 如何废除16位的中断机制？

先关中断（cli），然后将system模块从0x10000复制到0x0000处，覆盖掉了BIOS中断向量表和BIOS 数据区。

这样做，一箭三雕：

(1) 废除BIOS的中断向量表，即废除了实模式下的中断服务程序；

(2) 回收刚刚结束使用寿命的程序所占用的内存空间；

(3) 让内核代码占据内存物理地址最开始的、天然的、有利的位置。

1.1.9 实现32位寻址的动作是？

打开A20，这样cpu即可以进行32位寻址，最大的寻址空间是4GB。

1.1.10 如何切换处理器的工作方式？

CR0寄存器的第0位是PE(Protected Mode Enable 保护模式使能标志)，置一时，保护模式；置零时，实模式。

1.1.11 如何跳转入main函数中？

先将main函数的执行入口地址_main压栈，然后执行ret，pop出该地址到EIP中，即开始执行main函数了。

1.2 第二章：设备环境初始化及激活进程0

1.2.1 系统通过mem_map对1MB以上的内存分页进行管理，那为什么不对1MB以上的内存空间也采用同样的分页方法进行管理呢？

1MB以内是内核代码和只有由内核管理的大部分数据所在的内存空间，绝对不允许用户进程访问；1MB以上（特别是主内存区）主要是用户进程的代码、数据所在的内存空间。
所以，内核和用户进程的分页管理方法必须不同

- 对于内核，线性地址==物理地址；
- 对于用户进程，无法通过线性地址推算出物理地址。

这样，内核可以访问用户进程；而用户进程不能访问其他进程，更不能访问内核。
PS：嵌入汇编的方法：__asm__ (汇编代码)

1.2.2 硬件初始化的大体过程？

(1) 规划内存格局：

- 主内存区

- 缓冲区
- 虚拟盘（Makefile可配置）

(2) 设置及初始化缓冲区：free_list将buffer_head链接成双向循环链表。

(3) 设置及初始化虚拟盘：虚拟盘的末端是主内存的始端。

(4) 初始化mem_map：对内核和用户进程分别采用不同的分页管理方法。

(5) 初始化缓冲区管理结构：buffer_head在低地址端；缓冲块在高地址端。start_buffer=&end;这个end 是内核代码末端的地址，因为有可能动态加载内核模块导致内核末端地址不固定，所以在内核模块链接期间设置这个end值。

1.2.3 与外设有关的初始化过程？

(1) 设置根设备：即文件系统中的根目录，此时应该为装机时的linux .iso所在的光盘或硬盘。

(2) 初始化软盘、硬盘等。

1.2.4 为内核及用户进程的正确运行所做的初始化过程？

(1) 中断服务的挂接：串口、显示器、键盘等。

(2) 初始化进程0：

- 设置时钟中断，以便可以支持多进程轮流执行。
- 设置系统调用总入口（system_call），以便可以允许用户进程与内核交互。

1.2.5 如何激活进程0？

因为Linux OS规定，除了进程0以外，所有进程都要由一个已有进程在3特权级下创建。所以用防中断的方法将进程0的特权级由0翻转到3，实现激活进程0.

1.3 第三章：进程1的创建及执行

PS：C语言的sys_fork对应于汇编的_sys_fork。

1.3.1 进程0创建进程1的大体步骤？

(1) 进程0 fork()进程1；

(2) 在task[64] 中为进程1申请一个空闲位置并获取进程号；

(3) 调用copy_process 函数，以便使进程1 可以就绪；

(4) 设置进程1的分页管理；

(5) 进程1共享进程0的文件；

(6) 设置进程1在GDT中的表项；

(7) 使进程1处于就绪状态：$p->state = TASK\_RUNNING$;

1.3.2 内核第一次做进程调度，即从进程0到进程1的过程？

进程0执行到for(;;) pause(); 进入pause()之后执行到schedule(); 在schedule()函数中，先分析当前有没有必要进行进程切换，如果有必要，再进行具体的切换操作。

(1) 首先根据task[64]这个结构，第一次遍历所有的进程，只要地址指针不为空，就要针对它们的"alarm"和"signal"进行处理。In this case, 因为进程0此时并未收到任何信号，并且它的状态是"TASK_INTERRUPTIBLE"，不可能转变为"TASK_RUNNING"，所以这次遍历的处理过程没有具体效果。

(2) 第二次遍历所有的进程，比较进程的状态和时间片，找出处于就绪态且counter最大的进程。In this case, 因为目前只有进程0和1，只有进程1处于"TASK_RUNNING"，所以，执行switch_to(next)，切换到进程1 执行。

1.3.3 进程1如何加载根文件系统？

加载文件系统最重要的标志，就是把一个逻辑设备上的文件系统的根i节点，关联到另一个文件系统的i 节点上（即mount命令）。

有一个问题：别的文件系统可以挂在根文件系统上，那么根文件系统挂在哪里呢？**super_block[8]** 上。

所以，加载根文件系统最重要的标志，就是把根文件系统的根i节点挂在super_block[8]中根设备对应的超级块上。

具体过程为：

(1) 复制根设备的超级块到super_block[8] 中，将根设备中的根i节点挂在super_block[8]中对应根设备的超级块上。

(2) 将驻留缓冲区中16个缓冲块的根设备逻辑块位图、i节点位图分别挂接在super_block[8] 中根设备超级块的s_zmap[8]、s_imap[8] 上。

(3) 将进程1的pwd、root指针指向根设备的根i 节点。

1.4 第四章：进程2的创建以及执行

1.4.1 简述打开终端设备文件的过程？

(1) 通过open()函数创建标准输入设备文件：”/dev/tty0”

(2) 通过两次复制文件句柄(dup()函数)创建标准输出、标准错误输出设备文件

1.4.2 简述进程1创建进程2并切换到进程2执行的过程？

(1) 进程1调用fork，映射到sys_fork，调用find_empty_process()函数，为进程2寻找空闲的task，之后调用copy_process()复制进程；

(2) 进程2创建完毕后，fork返回，∵返回值是2，∴!($pid = fork()$)为假，∴调用wait()函数，wait()函数的功能是：

   (a) 如果进程1有等待退出的子进程，则为该子进程的退出做善后工作；

   (b) 如果进程1有子进程，但并不等待退出，则进行进程切换；

   (c) 如果没有子进程，函数返回。

   wait()函数最终会映射到系统调用函数sys_waitpid()中执行，该函数遍历所有的进程，以便确定哪个进程是进程1 的子进程，此时进程2即被选中了；再对进程2进行分析，∵进程2此时是就绪态，并不准备退出，∴设置flag标志为1，该标志将导致进程切换。

(3) 进入$if(flag)$ 去执行，内核先将进程1的状态设置为可中断等待状态，之后调用schedule()切换到进程2去执行。

1.4.3 简述加载shell程序的过程？

(1) 关闭标准输入设备文件，打开rc文件。
PS：rc文件是脚本文件，该文件记录着一些命令，应用程序通过解析这些命令来确定执行命令（run commands）。

(2) 检测shell文件，以便确定shell程序是否具备加载条件。

(3) 为shell程序的执行做准备。

- 加载参数和环境变量
- 调整进程2的管理结构
- 调整EIP和ESP：用shell程序的起始地址值设置EIP；用进程2新的栈顶地址值设置ESP，这样，软中断itet返回之后，进程2就将从shell程序开始执行。

(4) 执行shell程序。

- 通过“缺页中断”处理程序，加载第一页的shell程序
- 内核将该加载页映射到shell进程的线性地址空间中

1.4.4 update的进程号是？

∵shell进程的进程号是2，并且由shell进程通过”/etc/rc”脚本中读取”/etc/update &” 命令创建了update进程，∴update的进程号是3.
PS：update进程有一项很重要的任务：将缓冲区中的数据同步到外设上。每隔一段时间，update进程就会被唤醒，把数据往外设上同步一次，之后这个进程就会被挂起，即被设置为可中断等待状态，等待着下一次被唤醒后继续执行，如此周而复始。

1.4.5 简述shell的退出过程？

∵shell程序循环调用read()函数读取rc文件中的内容，∴总会有读取结束的时刻，此时返回值应为_ERROR，这个返回值将导致shell进程退出，给进程1发送SIGCHLD信号，导致切换到进程1执行。

1.4.6 简述重建shell以致实现系统怠速的过程？

进程1通过调用fork()函数重启shell进程（∴shell进程的PID依然是2），∵此次shell重新打开的是tty0文件而不是rc文件，∴此次shell开始执行之后不会再退出。
shell进程将被设置为可中断等待状态，这样所有的进程全部都处于可中断等待状态，再次切换到进程0去执行，系统实现怠速。怠速之后，用户即可通过shell进程与计算机进行交互了。

1.5 第五章：文件操作

1.5.1 简述安装文件系统的过程？(mount)

∵OS已经成功加载了根文件系统，∴OS能够以文件的形式与根设备进行数据交互。
PS：安装文件系统，就是在根文件系统的基础上，把硬盘中的文件系统安装在根文件系统上，使得OS也具备以文件的形式与硬盘进行数据交互的能力。在shell下输入“**mount /dev/hd1 /mnt**”来安装文件系统。
具体的过程如下：

- 获取外设的超级块
- 确定根文件系统的挂接点
- 将超级块与根文件系统挂接

1.5.2 简述打开文件的过程？(open)

首先要知道三个数据结构：

- *filp[20]：管理一个进程可以打开的文件；
- file_table[64]：管理所有进程可以打开的文件；
- inode_table[32]：管理正在使用的文件的i节点。 ***注意：在OS中，i节点和文件是一一对应的，找到i节点就能唯一确定文件。***

打开文件的关系示意图如图1所示：

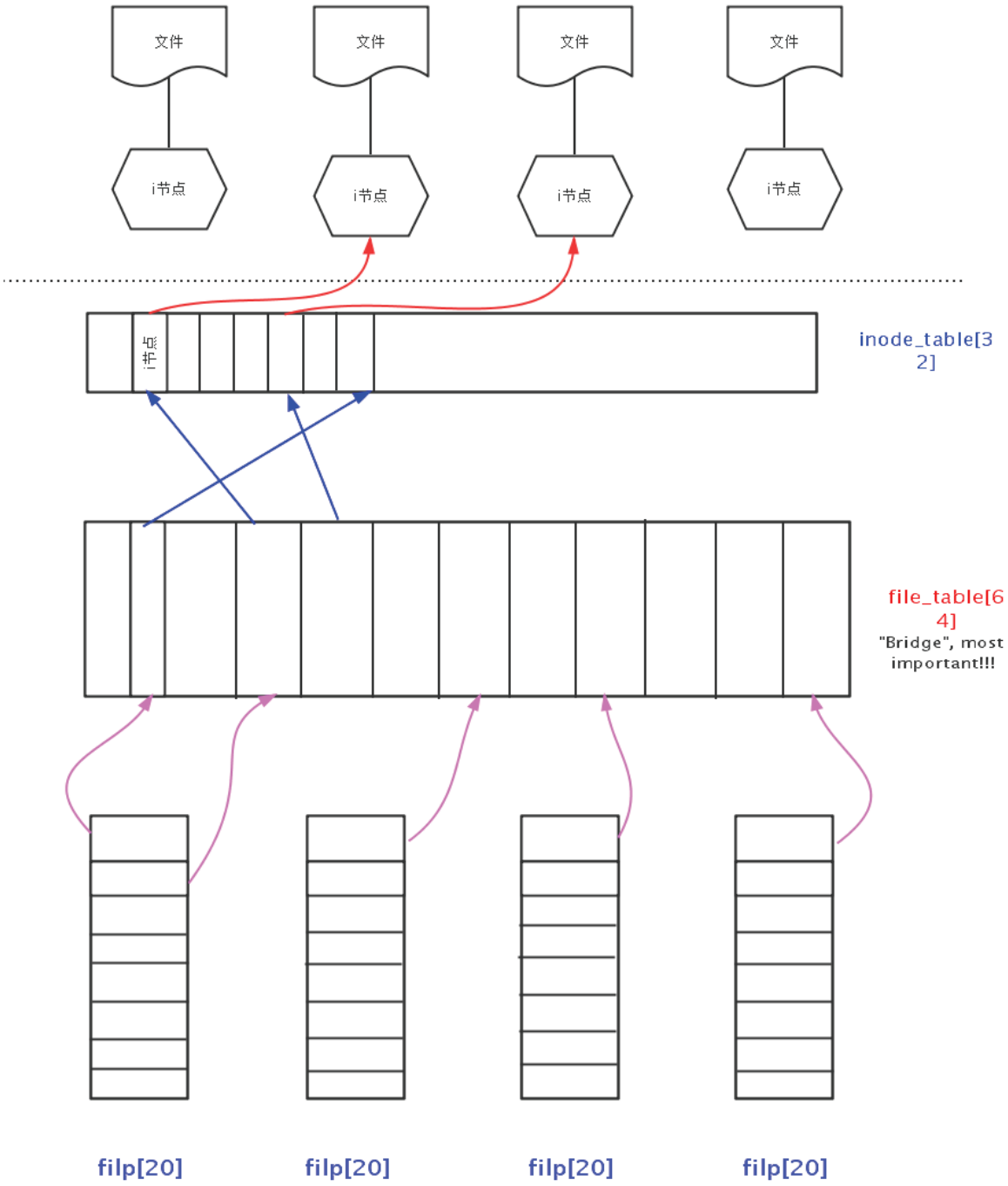

Figure 1: 打开文件的关系示意图。

打开文件需要三个步骤：

(1) 将进程的*filp[20]与file_table[64]挂接，目的是label某一个确定的用户进程；

(2) 获取文件i节点，载入到inode_table[32]中，目的是label该文件要被进行打开操作啦；

(3) 将inode_table[32]与file_table[64]挂接，此时file_table[64]中的挂接点，一端与当前进程的*filp[20]指针绑定，另一端与inode_table[32]中特定的文件的i节点绑定。

在绑定关系建立之后，OS把fd返回给用户进程，这个fd是挂接点在file_table[64]中的偏移量，即所谓的“文件句柄”。

1.5.3 简述读文件的过程？(read)

(1) 通过调用bmp()确定指定的文件数据块在外设上的逻辑块号；

(2) 通过调用bread()将数据块读入缓冲块中；

(3) 通过调用file_read()将缓冲块中的数据复制到进程空间中。

1.5.4 简述新建文件的过程？(create)

(1) 查找文件，∵是要被新建的文件，∴此时该文件并不存在，∴无法查找到该文件，bh值将被设为NULL。

(2) 新建文件i节点：new_inode()函数执行新建i节点的步骤分为两步：

  (a) 要在i节点位图中，对新建i节点对应的位予以标示；

  (b) 要将i节点的部分属性信息载入inode_table[32]表中指定的表项。

(3) 新建文件目录项：查找空目录项并添加目录数据。

1.5.5 简述写文件的过程？(write)

- 确定文件的写入位置：用户进程传递的flags参数，决定了文件的数据写入位置。一个新的数据块的具体创建工作包含两部分：

  (1) 将新建的数据块对应的逻辑块位图置一；

(2) 在缓冲区中为新建的数据块申请缓冲块，用以承载写入的内容。

- 申请缓冲块
- 将指定的数据从进程空间复制到缓冲块
- 将数据从缓冲区同步到外设：有两种情况：

  (1) update定期同步

  (2) 因缓冲区使用达到极限，OS就会强行同步

1.5.6 简述修改文件的过程？(write)

(1) 调用lseek()函数重定位文件的当前操作指针

(2) 修改文件

1.5.7 简述关闭文件的过程？(close)

(1) 当前进程的filp与file_table[64]脱钩

(2) 文件i节点被释放

2 《你必须知道的495个C语言问题》

2.1 第一章：声明和初始化

2.1.1 尽管unsigned char 型可以当成“小”整数使用，但这样做不值得，为什么？

因为编译器需要生成额外的代码来进行char型和int 型之间的转换，导致目标代码量变大；而且不可预知的符号扩展也会带来一些麻烦。

2.1.2 为什么不精确定义标准类型的大小？

对象的具体大小应该由具体的实现来决定。

PS：在C语言中，唯一能够让你以二进制位的方式指定大小的地方是结构中的位域。

2.1.3 这样声明：char *p1, p2; 代表什么？

在C语言中，声明的语法是：“基本类型 生成的基本类型的东西;”，所以，由上述代码，基本类型都是char型，而*p1表示生成的东西是一个指针；而p2表示生成的东西是一个普通变量。

PS：写成”char* ”是非常容易误导的，不要这样写！

2.1.4 怎样声明和定义全局变量和函数？

首先，

```
1 extern int i;
2 extern int f();
```

是声明；

```
1 int i = 0;
2 int f(){
3     return 0;
4 }
```

是定义。

即：全局变量的定义是真正分配空间（并赋初值）的声明；而全局函数的定义是提供函数体。

最好的方法就是：外部声明在.h文件中；定义在.c文件中。

2.1.5 静态变量和静态函数的规则中有一个细微的区别，是什么？

参照http://c-faq.com/decl/static.jd.html，如图2：

```
Example:

/* object */      /* function */

int o1;           int f1();           /* external linkage */
static int o2;    static int f2();    /* internal linkage */
static int o3;    static int f3();    /* internal linkage */

static int o1;    static int f1();    /* ERROR, both have external linkage */
int o2;                               /* ERROR, o2 has internal linkage */
                  int f2();           /* OK, picks up internal linkage */
extern int o3;    extern int f3();    /* OK, both pick up internal linkage */

The difference is case (2); where functions do pick up a previous linkage even without "extern", objects don't.
```

Figure 2: 静态变量和静态函数的规则中的细微区别。

为了便于我的理解，可以将internal linkage想象为恪守妇道的良家妇女属性，将external linkage想象为一入妓院深似海的东莞技工属性。先看第

一个案例，你已经成了婊子了，就别再想着试图从良了，没希望；然后看第二个案例，对于变量想当妓女，会失败，但是函数不同，说白了函数就是给多数人用的，当妓女是它的使命，所以，函数下海是可行的；最后看第三个案例，要是你们俩都有强烈的愿望（加了extern）要当妓女，好，那就全部成全你俩！

2.1.6 如何理解char *(*(*a[])()) (); 这句话？

可以使用typedef逐步完成声明：

```
1 typedef char *pc;
2 typedef pc fpc();
3 typedef fpc *pfpc;
4 typedef pfpc fpfpc();
5 typedef fpfpc *pfpfpc;
6 pfpfpc a[N];
```

2.1.7 C语言的三种连接类型是？

- 外部连接：全局、非静态变量和函数（在所有的源文件中有效）
- 内部连接：限于文件作用域内的静态函数和变量
- 无连接：局部变量及typedef名称和枚举变量

2.1.8 尚未初始化的初始值是？

- static：可以确保初始值为零
- automatic：垃圾内容
- malloc和realloc动态分配的内存包含垃圾内容
- calloc获得的内存全为零

2.1.9 如何初始化一个函数指针？

```
1 extern int func();
2 int (*pf)() = func;
```

当一个函数名(func)出现在上述表达式中时，它会“退化”成一个指针（即隐式地取出了它的地址）。

2.2 第二章：结构、联合和枚举

2.2.1 用struct x1 ...;声明之后如何定义？用typedef struct x2 ... x2;声明之后如何定义？

用struct x1 ...;声明之后如何定义？**struct x1 a;**
用typedef struct x2 ... x2; 声明之后如何定义？**x2 b;**

2.2.2 怎样从/向文件读/写一个结构？

```
1 fwrite(&somestruct, sizeof(struct somestruct), 1, filepointer);
2 fread(&somestruct, sizeof(struct somestruct), 1, filepointer);
```

2.2.3 为什么sizeof返回值有可能大于期望值？

因为在按字节寻址的机器中，2字节的short int 型变量必须放在偶地址上，4字节的long int 型变量必须放在4的整倍数地址上等，所以，编译器为了保证对齐要求，有可能会留出空洞，正是这些空洞的存在导致了sizeof返回值大于期望值。

2.2.4 如何求出一个域（f）在结构（s）中的字节偏移量？

使用offsetof(struct s, f)这个函数即可。深入下去：

```
1 #define offsetof(type, f) ((size_t) \
2     ((char *) &((type *)0 -> f) - (char *) (type *)0 ) );
```

总体思路还是：f的字节地址减去结构体的首地址。其中，转换成(char *)指针可以确保算出的偏移是字节偏移。

2.2.5 既然数组名可以用作数组的基地址，为什么对结构不能这样？

∵数组在c语言中处于“二级”状态，∴数组引用可以“退化”为指针。
但∵结构是一级对象，∴当你提到结构的时候，你得到的是整个结构。∴最好使用指针而不是直接使用结构。

2.2.6 结构和联合有什么区别？

联合本质上是一个成员相互重叠的结构，某一时刻只能使用一个成员。联合的大小是它最大成员的大小，而结构的大小是它所有成员大小之和。

2.2.7 如何自动跟踪联合的哪个域在使用？

可以自己实现一个显式带标签的联合：

```
struct tagged_union{
    enum{UNKNOWN, INT, LONG, DOUBLE, POINTER}code;
    union{
        int i;
        long l;
        double d;
        void *p;
    }u;
};
```

2.3 第三章：表达式

2.3.1 下面的代码可以不需要临时变量就能交换a和b的值，分析下？

```
    a ^= b ^=a ^= b;
```

待解决！！！

2.3.2 为何如下的代码不对？

```
int a = 1000, b = 1000;
long int c = a*b;
```

解答：这个乘法运算是用int进行的，其结果在提升给左侧的c之前已经溢出或被截断了，it's too late!
应为：

```
int a = 1000, b = 1000;
long int c = (long int) a*b;
```

2.3.3 “无符号保护”和“值保护”的区别？

待解决！！！

2.4 第四章：指针

2.4.1 已声明了一个指针：char *p; 那么，如何为它分配一些内存空间？如何为这些内存赋值？

```
1 char * p;
```

这一行代码声明的指针名字是p，而不是所谓的*p。

∵

- 当操作指针本身时，只需要使用指针的名字p即可；
- 当操作指针所指向的内存时，才需要使用*作为间接操作符。

∴

- 分配内存空间：p = malloc (10);
- 写内存：*p = 'H';

2.4.2 下面的代码错在哪里？

```
1 int array[5], i, *ip;
2 for(i=0; i<5; i++)
3     array[i]=i;
4 ip = array;
5 printf("%d\n", *(ip + 3 * sizeof(int)));
```

解答：∵在C语言中，指针算术自动采纳它所指向的对象的大小，
∴最后一行应改为：

```
1 printf("%d\n", *(ip + 3));
```

或者直接：

```
1 printf("%d\n", ip[3]);
```

2.4.3 c语言如何模拟“按引用传参”？

简单来讲就分为两步：

(1) 定义一个接受指针的函数；

(2) 为了调用该函数，在传参数的时候使用&操作符。

未完待续！！！

2.4.4 解释一下“通用对象指针”和“通用函数指针”？

首先，没有所谓的“通用指针类型”，但是∵当void * 和其他类型相互赋值时，如果需要，它可以自动转换为其他类型，又∵void * 指针只能保持对象（即数据啦！）指针，∴可以不严谨地将void * 称作“通用对象指针”。
∵所有的函数指针类型都可以相互转换，只要在调用之前再转回正确的类型即可，∴可以使用任何函数类型作为“通用函数指针”，但一般使用int(*) () 和void(*) () 这两种。

2.4.5 已有一个函数：extern f(int *); 它接受指向int型的指针。如何传入一个常量5？

使用“符合字面量”：

```
1 f((int []) {5});
```

2.4.6 根据如下代码，可知fp是一个函数指针。如何用指针调用该函数？

```
1 int f(), (*fp)();
2 fp = f;
```

解答：方法一：∵函数名在表达式和初始化中总是隐式地退化为指针，∴可以直接

```
1 fp();
```

但是这样无法保证在老编译器上的可移植性。所以最好采用方法二。
方法二：

```
1 (*fp) ();
```