

第二章 内核探索工具集

本章包括:

2.1 内核中常见的数据类型

2.2 汇编

2.3 汇编语言示例

2.4 内联汇编

2.5 特殊的C语言用法

2.6 内核探索工具一览

2.7 内核发言: 倾听来自内核的消息

2.8 其他 (**Miscellaneous Quirks**)

小结

项目: HelloMod

习题

本章简要介绍 Linux 中一般的编码结构，并描述诸多与内核打交道的方法。我们首先关注内核中常见数据类型，比如有效存储、信息检索、编码方法，以及基本汇编语言。这将为我们在以后的章节中详细分析内核打下基础。接下来介绍 Linux 如何将源代码编译、链接成可执行代码，这对理解交叉平台编码和更好地介绍 GNU 工具集不无益处。之后就是一系列从 Linux 内核搜集信息的方法之概要了。本章涉及的内容非常广泛，包括源代码和可执行代码的分析，以及在 Linux 内核中插入调试语句。最后，本章以“大杂烩”的形式对其他遇到的 Linux 习俗进行了观察和评论¹。

2.1 内核中常见的数据类型

Linux 中包含许多对象和数据结构，例如内存页面，进程和中断。如果操作系统要高效运行，那么如何及时地从多个对象中引用其中的一个对象是要关注的主要问题。Linux 使用链表和二叉搜索树（还有一组帮助例程）先将这些对象划分到不同的组，之后再以有效的方式在对应的组中查找单个元素。

2.1.1 链表

¹ 我们还没有深入研究内核内在的东西。由此看来，对于操纵整个内核代码而言，完全有必要对开发工具和基本概念进行概述。当然，如果你是一个有经验的内核爱好者，你可以跳过这一节，直接从第 3 章（“进程：程序执行的主要模型”）开始。

在计算机科学中，链表（linked lists）是常见的数据类型，并贯穿整个 Linux 内核的始终。在 Linux 内核中，链表常以循环双向链表的形式出现（参见图 2.1）。因此，给定链表中的任一结点，均可找到其下一结点和前一结点。有关链表定义的所有代码可以查看头文件 `include/linux/list.h`。本节讨论链表的主要特征。

图 2.1 调用宏 `INIT_LIST_HEAD` 后的链表

使用宏 `LIST_HEAD` 和 `INIT_LIST_HEAD` 初始化链表：

```
-----  
-----  
include/linux/list.h  
  
27  
  
28 struct list_head {  
  
29     struct list_head *next, *prev;  
  
30 };  
  
31  
  
32 #define LIST_HEAD_INIT(name) { &(amp;name), &(amp;name) }  
  
33  
  
34 #define LIST_HEAD(name) \  
  
35     struct list_head name = LIST_HEAD_INIT(name)  
  
36  
  
37 #define INIT_LIST_HEAD(ptr) do { \  

```

```
38 (ptr)->next = (ptr); (ptr)->prev = (ptr); \
39 } while (0)
```


34 行:

宏 LIST_HEAD 根据给定的名字 name 创建链表的表头

37 行:

宏 INIT_LIST_HEAD 初始化表头结点中的 prev 指针和 next 指针，完成之后，name 就是一个空的双向链表²。

相应地，简单栈和队列可以由函数 list_add () 和 list_add_tail () 来实现，工作队列的代码中有一个不错的例子:

kernel/workqueue.c

```
330 list_add(&wq->list, &workqueues);
```


内核将 wq->list 加入到系统的工作队列链表——workqueues 中，因此 workqueues 就是一组队列。

与之类似，下列代码将 work->entry 加入到 cwq->worklist 的末尾，cwq->worklist

因而也被当作队列:

² 一个空的链表被定义成：其头结点的 next 指针指向该链表的表头本身

kernel/workqueue.c

```
84 list_add_tail(&work->entry, &cwq->worklist);
```

使用 `list_del ()` 可以从链表中删除元素。`list_del ()` 将链表元素作为参数，删除

元素时，仅需修改该元素的下一结点和前一结点的指针，使之互相指向对方即可。例如，

当撤销一个工作队列时，下列代码可以从系统的工作队列链表中删除该工作队列：

kernel/workqueue.c

```
382 list_del(&wq->list);
```

`include/linux/list.h` 中定义了一个特别有用的宏 `list_for_each_entry`：

`include/linux/list.h`

```
349 /**
```

```
350 * list_for_each_entry - iterate over list of given type
```

```
351 * @pos: the type * to use as a loop counter.
```

```
352 * @head: the head for your list.
```

```

353 * @member: the name of the list_struct within the struct.
354 */
355 #define list_for_each_entry(pos, head, member)
356     for (pos = list_entry((head)->next, typeof(*pos), member),
357         prefetch(pos->member.next);
358         &pos->member != (head);
359         pos = list_entry(pos->member.next, typeof(*pos), member),
360         prefetch(pos->member.next))

```


该函数循环遍历整个链表，对链表中的每个元素都起作用。举个例子：CPU 工作时，

将从每个工作队列中唤醒一个进程：

kernel/workqueue.c

```

59 struct workqueue_struct {
60     struct cpu_workqueue_struct cpu_wq[NR_CPUS];
61     const char *name;
62     struct list_head list; /* Empty if single thread */
63 };
...
466 case CPU_ONLINE:

```

```

467     /* Kick off worker threads. */
468     list_for_each_entry(wq, &workqueues, list)
469         wake_up_process(wq->cpu_wq[hotcpu].thread);
470     break;

```


该宏在 `workqueue_struct wq` 中扩展并应用 `list_head` 链表，以遍历头指针指向工作队列的链表。如果这看起来有点混淆的话，请记住，我们并不需要为了遍历而知道这究竟是哪个链表中的结点。直到当前结点的 `next` 指针值等于该链表的头结点³时，我们就访问到该链表的表尾。有关工作队列的说明参见图 2.2。

图 2.2 工作队列链表

链表的改进使得头结点中仅有一个指向第一个元素的指针，刚好与在前一节中讨论过的带有双指针的头结点形成对比。仅有一个指针的头结点应用于哈希表（参见第四章，“内存管理”），它没有指向链表表尾元素的指针。由于在哈希查找中不常用到尾指针，因而这样做可以节省内存空间。


```
include/linux/list.h
```

³ 我们也可以通过 `list_for_each_entry_reverse()` 来反向遍历链表。

```
484 struct hlist_head {
485     struct hlist_node *first;
486 };

488 struct hlist_node {
489     struct hlist_node *next, **pprev;
490 };

492 #define HLIST_HEAD_INIT { .first = NULL }
493 #define HLIST_HEAD(name) struct hlist_head name = { .first = NULL }
```


492 行:

宏 `HLIST_HEAD_INIT` 将指针 `first` 置为空指针 (null pointer)。

493 行:

宏 `HLIST_HEAD` 根据 `name` 创建链表，并将指针 `first` 置为空指针。

在整个 Linux 内核代码的工作队列中都用到这些链表结构，我们将会看到在调度程序、定时器、模块处理例程中看到。

2.1.2 查找

上一节我们探究了链表中的分组元素。元素的有序表根据每个元素的关键值进行排序（比如说，每个元素的关键值均大于其前一元素中对应的值）。如果想要找到某个特定元素的话（基于其关键值），我们可以从表头开始，顺序查找整个链表，将当前结点的关键值与给定的关键值比较。若不相等，则继续比较下一个元素，直到找到匹配的值。在这个例子中，从链表中查找给定元素所花的时间与关键值成正比。换句话说，若增加链表中的元素个数，这种线性查找将花费更多时间。

大O表示法:

对于查找算法而言，大O表示法常用于从理论上来衡量一个算法找到某给定关键值的执行效率，它代表对于一个给定值n在最坏情况下所花费的查找时间。线性查找的效率是 $O(n/2)$ ，这就意味着，找到一个给定关键值平均必须查找半个链表。

出处：美国标准技术协会 (www.nist.gov)

对于元素较多的链表而言，为了不使操作系统空等，需要更快地存储和定位给定的数据。虽然目前已经有很多方法（包括其衍生方法），Linux存储数据时用的另一主要数据结构还是树。

2.1.3 树

树用于 Linux 内存管理中，能够有效地访问并操作数据。此时，衡量其有效性就是看存储及从多个数据中检索单个数据的速度有多快。本节将讨论基本树，尤其是红黑树，而关于树在 Linux 下的实现及帮助例程，请参考第六章“文件系统”。在计算机科学中，有根树由结点和边组成（见图 2.3），结点代表数据元素，边代表结点之间的路径，第一个结点，或者说顶层结点，就是树的根结点。结点之间的关系有双亲、孩子、兄弟这三种。每个孩子结点有且仅有一个双亲结点（树根除外），每个双亲结点可以有一个或多个孩子结点，互为兄弟的结点有共同的双亲，没有孩子的结点称之为叶结点。树的高度是指从树根到最远的叶结点之间的边数。树的每一行子孙被称之为层。图 2.3 中，b 和 c 位于 a 的下一层，而 d、e、f 位于 a 下面两层。查找给定兄弟集合中的某一数据元素时，有序树最左边的兄弟其值最小，而最右边的兄弟其值最大。树通常以链表和数组的形式实现，而在树中移动的过程就叫树的遍历。

图 2.3 有根树

2.1.3.1 二叉树

以前，我们用线性查找的方式来查找关键值，在每次循环中比较关键值。每次比较我们可以减少有序表中一半的结点吗？

二叉树和链表不同，它是一种分层的数据结构，而不是线性的。在二叉树中，每个元素或结点指向一个左孩子或右孩子结点，每个孩子结点又指向一个左孩子或右孩子，依此类推。其结点之间排序的主要规则就是左孩子的关键值小于双亲，而右孩子的关键值大于或等于双亲。因此，对于一个给定结点的键值，其左子树上所有结点的键值均小于该结点，而其右子树上所有结点的键值均大于或等于该结点。

往二叉树中存放数据时，首先必须找到适当的插入位置，而每次循环均可减少一半要查找的数据个数。用大O表示法来表示时，其性能（关于查找的次数）就是 $O \log(n)$ ，相比之下，线性查找的性能是 $O(n/2)$ 。

遍历二叉树的算法比较简单。对于每个结点而言，比较完该结点的键值后，就可以遍历其左子树或右子树，因而，二叉树的遍历本身就很方便用递归来实现。下面将讨论其具体实现，辅助函数以及二叉树的类型。

刚才我们提到，二叉树中的结点可以有一个左孩子，或者一个右孩子，或者有左、右两个孩子，也可以没有孩子。二叉有序树的规则是，给定一个结点的值 x ，其左孩子（包

括所有子孙结点) 的值小于 x , 而其右孩子 (包括所有子孙结点) 的值大于 x 。由此可知, 如果将数据的有序集合插入到二叉树中, 将形成一个线性列表, 对于一个给定值, 其查找速度就会变得和线性查找一样慢。例如, 根据数据集[0,1,2,3,4,5,6]创建一颗二叉树时, 0 是树根; 1 比 0 大, 是 0 的右孩子; 2 比 1 大, 是 1 的右孩子; 而 3 是 2 的右孩子; 依此类推。

在均高二叉树中, 根到任意叶结点的距离都是最远的。结点添加到二叉树中后, 为了保证查找的效率, 必须进行平衡化处理, 这可以通过旋转来实现。插入一个结点后, 给定结点 e , 如果它有一个比任何其他叶结点高两层的左子树的话, 就必须对 e 作右旋转。如图 2.4 所示, e 变成 h 的双亲, e 的右孩子则变成 h 的左孩子。若每次插入结点后都进行了平衡化处理, 我们最多只需作一次旋转。满足平衡规则 (若某结点的孩子结点是一个叶结点的话, 它们之间的间距不会超过 1) 的二叉树被称为 AVL 树 (这一术语最初是由 G.M.Adelson-Velskii 和 E.M.Landis 提出来的)。

图 2.4 二叉树的右旋转

2.1.3.2 红黑树

红黑树类似于 AVL 树，用于 Linux 内存管理。红黑树就是二叉平衡树，其每个结点都有红或黑的颜色属性。

红黑树的规则如下：

- 任何结点要么是红的，要么是黑的。
- 如果一个结点是红的，那么它的所有孩子都是黑的；
- 所有叶结点都是黑的。
- 从根往叶结点遍历时，每条路径包含同样多的黑结点。

AVL 树和红黑树的查找效率都是 $O \log(n)$ ，而根据插入（已排序的/未排序的）和查找数据的不同，每次都能得出不同的具体数值（网上有一些讨论二叉查找树[BSTs]性能的文章，有兴趣的话，可以找来看看）。

前面已经提到，许多其他数据结构和相关的查找算法也被应用于计算机科学中。本节的主要目的是，希望通过介绍 Linux 中常用数据结构的基本概念，有助于你的探索。对链表和树结构有了基本的理解后，可以更好地理解复杂的操作，例如我们将在后续章节中来讨论的内存管理和队列。

2.2 汇编

作为一个操作系统，Linux 的某些部分不可避免地处理器密切相关。当然，Linux 设计者们已做了大量工作，使得处理器（或“体系结构”）相关的代码尽可能少一些，争取在所有支持的体系结构中重用的代码尽可能多一些。本节探讨如下内容：

- 同一 C 函数在 x86 和 PowerPC 体系结构中如何实现。
- 宏与内联汇编代码的应用。

本节旨在提供足够的基本知识，以便你研究特定体系结构的内核代码时，能很好地理解它，而不至于手足无措。有关汇编语言程序设计的更多内容，还是留给其它书籍吧。

此外，我们还将介绍一些与体系结构相关的最复杂的代码：内联汇编。

讨论 PPC 与 x86 汇编语言之前，还是先来看看这两种体系结构吧。

2.2.1 PowerPC

PowerPC 是精简指令集计算机（Reduced Instruction Set Computing, RISC）体系结构。RISC 希望能在尽可能少的处理器周期中执行简单指令集，以提高系统性能。我们很快就会看到，由于利用了硬件可执行并行指令（超标量体系结构）的特性，某些指令其实

非常复杂。IBM、Motorola、Apple 公司联合定义了 PowerPC 体系结构。表 2.1 列出了 PowerPC

中可供用户使用的寄存器。

表 2.1 PowerPC 的用户寄存器集合

寄存器名	不同体系结构下的长度		功能	寄存器数量
	32 位	64 位		
CR	32	32	条件寄存器	1
LR	32	64	链接寄存器	1
CTR	32	64	计数寄存器	1
GPR[0..31]	32	64	通用寄存器	32
XER	32	64	定点异常寄存器	1
FPR[0..31]	64	64	浮点寄存器	32
FPSCR	32	64	浮点状态控制寄存器	1

表 2.2 说明了通用寄存器和浮点寄存器在应用二进制接口方面的应用。可变寄存器任

何时候都可以使用，专用寄存器用于特定场合，非可变寄存器使用时必须在整个函数调

用过程中加以保护。

表 2.2 应用二进制接口寄存器的用法

寄存器	类型	使用
r0	Volatile	序言/结尾, 与具体语言相关
r1	Dedicated	栈指针
r2	Dedicated	TOC
r3-r4	Volatile	参数传递, 输入/输出
r5-r10	Volatile	参数传递
r11	Volatile	环境指针

r12	Volatile	异常处理
r13	Non-volatile	必须在调用过程中加以保护
r14-r31	Non-volatile	必须在调用过程中加以保护
f0	Volatile	擦除
f1	Volatile	第一个 FP 参数, 第一个 FP 标量返回值
f2-f4	Volatile	第二到第四个参数, FP 标量返回值
f5-f13	Volatile	第五到第十三个参数
f14-f31	Non-volatile	必须在调用过程中加以保护

应用二进制接口 (ABI)

ABI 是一些规范的集合, 例如: 调用规则、机器接口、操作系统接口等。它允许链接程序将已编译好的单个模块链接成一个整体, 而不需要重新编译。ABI 定义了这些单元之间的二进制接口。目前, PowerPC 有几个 ABI 的变体, 它们通常与目标操作系统和硬件相关。这些变体有基于 UNIX System V 应用二进制接口的文档, 最初源自 AT&T, 后来又根据 SCO 修改而成。遵循 ABI 的好处在于: 可以链接不同编译程序所编译的目标文件。

32 位 PowerPC 体系结构采用 4 个字节长的指令, 并且按字对齐, 存取时按字节、半字、

字、双字来操作, 可分为分支指令、定点指令和浮点指令这几类。

2.2.1.1 分支指令

条件寄存器 CR (condition register) 对所有分支操作而言都是一个整体, 它可以划分为 8 个 4 位的区域, 其值可由 move 指令显式设置, 也可由某条指令隐式设置, 最常见的是由比较指令隐式设置。

链接寄存器 LR (link register) 由特定的分支指令使用, 在某分支执行完后提供目标地址和返回地址。

计数寄存器 CTR (count register) 由特殊的分支指令循环执行减 1 计数的操作。CTR 和某些特定的分支指令配合使用时, 还可保存目标地址。

除上述 CTR 和 LR 外, PowerPC 的分支指令还可以跳转到某一相对或绝对地址, 通过使用扩展的助记手段, 可以产生多种形式的条件转移和无条件转移。

2.2.1.2 定点指令

PPC 中没有修改存储器的计算指令, 所有工作都必须从一个或多个通用寄存器 (GPR) 开始。在高位地址优先的排序法中, 存储器存取指令按字节、半字、字、双字来存取数据。使用扩展的助记手段后, 增加了许多加载、存储、运算和逻辑的定点指令, 以及移入/移出系统寄存器的特殊指令。

2.2.1.3 浮点指令

浮点指令可分为两类：一类是计算指令，包括运算指令、取整指令、转换指令和比较指令；另一类是非计算指令，包括移入/移出存储器或另一寄存器的指令。系统中共有 32 个通用浮点寄存器，均可用于存放双精度浮点数。

高位地址优先/低位地址优先

在处理器体系结构中，Endianness 是指字节顺序和操作。PowerPC 采用高位地址优先的方式，就是说，最高位字节存储在低地址空间，最低位字节存储在随后的 3 个字节中（对 32 位数据而言）。低位地址优先用于 x86 体系结构，与高位地址优先刚好相反，最低位字节位于低位地址空间，而最高位字节位于随后的 3 个字节中。让我们来看看 0x12345678 的表示（见图 2.5）

图 2.5 高位地址优先和低位地址优先排序

本书并不讨论哪个系统采用的方式更好一些，但编写和调试代码时，了解你是在哪种系统上工作是极为重要的。这儿有一个错用 Endianness 的例子：基于一种体系结构编写但用于另一体系结构的 PCI 设备驱动程序。

术语高位地址优先和低位地址优先源自 Jonathan Swift 撰写的《格利佛游记》。在这个故事中，格利佛发现了吃熟鸡蛋的方式完全不同的两个民族，一个从大端开始，另一个从小端开始。

2.2.2 x86

x86 体系结构是一种复杂指令集计算机（Complex Instruction Set Computing, CISC）体系结构。根据其作用的不同，指令的长度是可变的。奔腾系列的 x86 体系结构有三种寄存器：通用寄存器、段寄存器和状态/控制寄存器。其基本用户集如下。

8 个通用寄存器及其用法如下所示：

EAX: 通用累加器

EBX: 指向数据的指针

ECX: 用于循环操作的计数器

EDX: I/O 指针

ESI: DS 段中指向数据的指针

EDI: ES 段中指向数据的指针

ESP: 堆栈指针

EBP: 栈中指向数据的指针

以下 6 个段寄存器用于实模式中的寻址操作。此时按块访问内存，因而内存中某个给定的字节可通过该段中的偏移量来引用（例如，ES: EDI 引用的是 ES（附加段）中由偏移量 EDI 表示的单元）：

CS: 代码段

SS: 堆栈段

ES、DS、FS、GS: 数据段

EFLAGS 寄存器存放每条指令执行完后处理器的状态，它可以保存诸如 0、溢出（overflow）、进位（carry）之类的结果。EIP 是专用的指针寄存器，存储处理器当前指令的偏移量，常用于代码段寄存器，以形成一个完整的地址（如，CS: EIP）。

EFLAGS: 状态、控制字，系统标志

EIP: 指令指针，包含 CS 段中的偏移量

x86 体系结构中的数据顺序遵循低位地址优先的方式，访问内存时可按字节（8 位）、字（16 位）、双字（32 位）和四字（64 位）访问。地址转换（及其相关的寄存器）见第四

章，本节中只需知道代码中常用的寄存器就足够了。x86 体系结构中有关数据的操作指令可分为控制指令（control）、算术运算指令（arithmetic）和数据操作指令（data）三类。

2.2.2.1 控制指令

与 PPC 中的分支指令类似，控制指令可以改变程序的流程。x86 体系结构使用了各种各样的 jump 指令和标号，使之能够根据 EFLAGS 寄存器的值选择执行的代码。jump 指令有多种变体，表 2.3 列举出了其中最常用的一些变体。条件代码要根据特定指令的结果来设置。例如，对两个整数执行 cmp 指令后，将修改 EFLAGS 寄存器的如下标志位：OF（溢出）、SF（非负标志）、ZF（零标志）、PF（奇偶位）、CF（进位标志）。因此，若 cmp 指令比较的是两个相等的操作数，零标志将被置位。

表 2.3 常见的 jump 指令

指令	作用	EFLAGS 条件代码
je	相等时跳转	ZF=1
jg	大于时跳转	ZF=0, SF=OF
jge	大于等于时跳转	SF=OF
j1	小于时跳转	SF! =OF
jle	小于等于时跳转	ZF=1
jmp	无条件跳转	没有条件

在 x86 汇编代码中，标号由唯一的名字后加冒号组成。它可以出现在汇编程序的任何地方，并与紧跟其后的那行代码具有相同的地址。下列代码中使用了条件跳转和标号：

100 pop eax

101 loop2:

102 pop ebx

103 cmp eax, ebx

104 jge loop2

100 行：

取出栈顶元素并将其值存入 `eax` 中。

101 行：

名为 `loop2` 的标号。

102 行:

取出栈顶元素并将其值存入 `ebx` 中。

103 行:

比较 `eax` 与 `ebx` 的值。

104 行:

`eax` 的值大于等于 `ebx` 的值时跳转。

另一种控制程序转移的方法是使用 `call` 和 `ret` 指令。参见下面这行汇编代

码: -----

```
call my_routine
```

这条 `call` 指令控制程序转移到 `my_routine` 处，同时将其下一条指令的地址压入堆栈，

然后，`ret` 指令（在 `my_routine` 中执行）将弹出返回地址，并跳转到该位置。

2.2.2.2 算术运算指令

流行的算术运算指令有 `add`、`sub`、`imul`（整数乘）、`idiv`（整数除），以及 `and`、`or`、`not`、`xor` 等逻辑操作。

本书并不讨论 x86 的浮点指令及其相关寄存器。近来，Intel 和 AMD 体系结构的扩展，诸如 MMX、SSE、3DNow、SIMD、SSE2/3 等，大大增强了数学协处理器的应用，例如图形和声音方面的应用。相关体系结构其详情可查阅编程手册。

2.2.2.3 数据操作指令

数据可以在寄存器之间、寄存器和内存之间传递，也可将常数存入寄存器或内存，但不能从内存直接传到内存。请看如下示例：

```
100  mov  eax,ebx

101  mov  eax,WORD PTR[data3]

102  mov  BYTE PTR[char1],a1

103  mov  eax,0xbeef
```

104 mov WORD PTR [my_data],0xbeef

100 行:

将 32 位数据从 ebx 传入 eax。

101 行:

将 32 位数据从内存变量 data3 传入 eax。

102 行:

将 8 位数据从内存变量 char1 传入 al。

103 行:

将常数 0xbeef 传入 eax。

104 行:

将常数 0xbeef 传入内存变量 my_data。

如上例所示, push、pop、pushl、popl 可以将数据移入/移出堆栈 (由 SS: ESP 来定位)。与 mov 指令类似, push 和 pop 操作也可用于寄存器、数据和常数。

2.3 汇编语言示例

我们可以创建一个简单的程序来看看对于同样的 C 语言代码，不同体系结构下究竟是怎样来生成汇编语言代码的。本次实验在 Red Hat 9 下使用 gcc 编译程序，而在 PowerPC 中使用 gcc 交叉编译程序。我们给出了 C 程序，并对照给出 x86 下和 PowerPC 下的汇编代码。

看到寥寥几行 C 语言代码产生了多少汇编代码后，你也许会大吃一惊吧。我们仅将 C 语言代码编译成汇编语言代码，没有链接任何环境代码，即使是 C 运行库和创建/销毁局部堆栈的代码也没有链接进来，因此其大小当然比实际的 ELF 可执行程序小很多。

值得注意的是，在汇编程序中，你几乎可以确切地看到处理器在周而复始地运行过程中取了什么指令。当然，你也可以完全控制你的代码和系统。值得一提的是：即使指令是从内存中顺序取出来的，它们的执行顺序也不是每次都和读入顺序完全一样，某些体系结构中，顺序加载和存储操作是相互独立的。

以下是该例子的 C 语言代码：

count.c

```
1 int main()
```

```
2 {
```

```
3  int i,j=0;
```

```
4
```

```
5  for(i=0;i<8;i++)
```

```
6    j=j+i;
```

```
7
```

```
8  return 0;
```

```
9 }
```

1行:

main 函数的定义。

3行:

将局部变量 i 、 j 初始化为 0。

5 行:

for 循环: 当 i 从 0 加到 7 时, 使得 $j = j + i$ 。

8 行:

return 表示跳转回调用程序。

2.3.1 x86 中的汇编示例

这是 x86 中在命令行内键入 `gcc S count.c` 后产生的代码。阅读代码前, 应当知道堆栈的基址由 `SS: EBP` 给出; 代码是以“AT&T”格式生成的, 寄存器前要加 `%`, 常数前加 `$`。读完本节提供的前述汇编指令样例后, 你应该能读懂这个简单的程序了, 但进一步研究之前, 还要讨论一个间接寻址的变量。

当涉及到内存中的某个位置时 (比如说: 栈), 汇编程序用特殊的语法来表示索引寻址。基址寄存器放在圆括号内, 而索引 (或偏移量) 放在圆括号外, 有效地址就是索引值加上基址寄存器的值。例如, 若 `%ebp` 的值是 20, 则 `8 (%ebp)` 的有效地址就是 $(8) + (20) = 12$

count.s

1 .file "count.c"

2 .version "01.01"

3 gcc2_compiled:

4 .text

5 .align 4

6 .globl main

7 .type main,@function

8 main:

 # 为 i 和 j 创建 8 字节的局部内存区

9 pushl %ebp

10 movl %esp, %ebp

11 subl \$8, %esp

将 i (ebp-4) 和 j (ebp-8) 初始化为 0

12 movl \$0, -8(%ebp)

13 movl \$0, -4(%ebp)

14 .p2align 2

15 .L3:

这是一个 for 循环的测试部分

16 cmpl \$7, -4(%ebp)

17 jle .L6

18 jmp .L4

19 .p2align 2

20 .L6:

这是 for 循环的循环体

21 movl -4(%ebp), %eax

22 leal -8(%ebp), %edx

23 addl %eax, (%edx)

```
24 leal -4(%ebp), %eax
```

```
25 incl (%eax)
```

```
26 jmp .L3
```

```
27 .p2align 2
```

```
28 .L4:
```

```
    # 设置该函数的退出代码
```

```
29 movl $0, %eax    30 leave    31 ret
```

9行:

将栈基址指针压入堆栈。

10行:

将栈指针传给基址指针。

11行:

从ebp开始，获取8个字节的堆栈内存。

12 行:

将 0 存入地址 `ebp8 (j)` 。

13 行:

将 0 存入地址 `ebp4 (i)` 。

14 行:

这是一个汇编命令，表明该指令是半字对齐的。

15 行:

这是由汇编程序创建的标号，名为 `.L3`。

16 行:

该指令比较 `i` 和 `7`。

17 行:

如果 `4 (%ebp)` 的值小于等于 `7`，则跳转到标号 `.L6`。

18 行:

否则，跳转到 `.L4`。

19 行:

对齐。

20 行:

标号.L6。

21 行:

将 i 的值传给 eax。

22 行:

将 j 的地址加载到 edx。

23 行:

将 i 的值加到由 edx (j) 指定的地址。

24 行:

将 i 的新值传给 eax。

25 行:

i 加 1。

26 行:

跳转，回到 for 循环中进行测试。

27 行:

按照 14 行的注释所描述的方式对齐。

28 行:

标号.L4。

29 行:

在 `eax` 中设置返回代码。

30 行:

释放局部内存区。

31 行:

从栈中弹出变量的值和返回地址，跳转回调用程序。

2.3.2 PowerPC 中的汇编示例

以下是根据 C 程序产生的 PPC 汇编代码。如果你熟悉汇编语言（及其简写），对许多 PPC 指令的功能就十分清楚了。但是，我们还得讨论几个基本指令的派生形式：

- `stwu RS, D (RA)` (Store Word with Update)：这一指令取出通用寄存器

(GPR) RS 的值，并存储到形如 $RA + D$ 的有效地址空间。之后，用新的有效地址来更新通用寄存器 (GPR) RA。

- `li RT, RS, SI` (Load Immediate) : 这是定点装入指令的扩展，用于将 RT, RS, SI 的值相加，其中通用寄存器 RS 和 SI 的和存储于 RT 中，RS 和 SI 是两个 16 位补码整数。若 RS 是 (GPR) R0，则 SI 的值存储在 RT 中。要注意的是，仅 16 位的数值必须作如下操作，即 opcode、寄存器和数值都必须重新编码成长度为 32 位的指令。
- `lwz RT, D (RA)` (Load Word and Zero) : 这条指令和 `stwu` 一样，形成一个有效地址，并从内存中加载一个字的数据到通用寄存器 RT 中。当 64 位地址运行于 32 位模式时，“零”是指生成的有效地址其前 32 位被置为 0。（详情参见“PowerPC 体系结构 I”一书)
- `blr` (Branch to Link Register) : 这条指令无条件转移到存储在链接寄存器中的 32 位地址。调用函数时，调用程序将返回地址存入链接寄存器。与 x86 中的 `ret` 指令类似，`blr` 是从函数返回的常用方式。

以下代码是从命令行键入 `gcc S count.c` 后生成的:

countppc.s

1 .file "count.c"

2 .section ".text"

3 .align 2

4 .globl main

5 .type main,@function

6 main:

 #从栈空间创建 32 字节的内存区，并初始化 i 和 j。

7 stwu 1,-32(1) #Store stack ptr (r1) 32 bytes into the stack

8 stw 31,28(1) #Store word r31 into lower end of memory area

9 mr 31,1 #Move contents of r1 into r31

10 li 0,0 #Load 0 into r0

11 stw 0,12(31) #Store word r0 into effective address 12 (r31) , var j

```
12 li 0,0    #load o into r0

13 stw 0,8(31) #Store word r0 into 8 (r31) , var i

14 .L2:

    #for 循环的测试部分

15 lwz 0,8(31) #Load i into r0

16 cmpwi 0,0,7 #compare word immediate r0 with integer value 7

17 ble 0,.L5 #Branch if less than or equal to label .L5

18 b .L3    #Branch unconditional to label .L3

19 .L5:

    #for 循环的循环体

20 lwz 9,12(31) #Load j into r9

21 lwz 0,8(31) #Load i into r0

22 add 0,9,0 #Add r0 to r9 and put result in r0

23 stw 0,12(31) #Store r0 into j

24 lwz 9,8(31) #load i into r9
```

25 addi 0,9,1 #Add 1 to r9 and store in r0

26 stw 0,8(31) #Store r0 into i

27 b .L2

28 .L3:

29 li 0,0 #Load 0 into r0

30 mr 3,0 #move r0 to r3

31 lwz 11,0(1) #load r1 into r11

32 lwz 31,-4(11) #Restore r31

33 mr 1,11 #Restore r1

34 blr #Branch to Link Register contents

-

7行:

将 stack ptr (r1) 的32个字节存入堆栈。

8行:

将 r31 存储到内存区的低地址空间。

9 行:

将 r1 的值传入 r31。

10 行:

将 0 加载到 r0。

11 行:

将 r0 存储到有效地址 12 (r31) 即变量 j 中。

12 行:

将 0 加载到 r0。

13 行:

将 r0 存储到有效地址 8 (r31) 即变量 i 中。

14 行:

标号 L2。

15 行:

将 i 加载到 r0。

16 行:

比较立即数 r0 与整数 7。

17 行:

r0 小于等于 7 时跳转到标号 L5。

18 行:

无条件跳转到标号 L3。

19 行:

标号 L5。

20 行:

将 j 加载到 r9。

21 行:

将 i 加载到 r0。

22 行:

r0 加上 r9，其结果存储在 r0 中。

23 行:

将 r0 的值存储到变量 j 中。

24 行:

将 i 加载到 r9。

25 行:

r9 加 1, 其结果存储在 r0 中。

26 行:

将 r0 的值存储到变量 i 中。

27 行:

这是无条件跳转分支 L2。

28 行:

标号 L3。

29 行:

将 0 加载到 r0。

30 行:

将 r0 的值传给 r3。

31 行:

将 r1 的值加载到 r11。

32 行:

恢复 r31 的值。

33 行:

恢复 r1 的值。

34 行:

跳转到链接寄存器所存储的地址处。

对照这两个汇编程序文件，发现它们的行数几乎是一样的，进一步观察可以看出，

RISC (PPC) 处理器使用了许多加载和存储指令，而 CISC (x86) 更偏好 mov 指令。

2.4 内联汇编

gcc 编译程序支持的另一种编码形式是内联汇编 (inline assembly) 代码。名如其言，

内联汇编不需要调用单独编译的汇编程序。我们可以通过特定的结构告诉编译程序将代码

块组合到一起，而不是要编译该代码块。虽然这样做会生成体系结构相关的文件，但能大

大提高 C 函数的可读性和执行效率。

以下就是内联汇编程序的结构：

```
1 asm (assembler instruction(s)
2   : output operands (optional)
3   : input operands (optional)
4   : clobbered registers (optional)
5 );
```

例如，内联汇编程序最基本的形式是：

```
asm ("movl %eax, %ebx");
```

也可以写成

```
asm ("movl %eax, %ebx" : : : );
```

由于确实要修改 `ebx` 的内容，我们逐渐揭开了编译程序神秘的面纱。

接受并修改 C 表达式，将它们返回给程序，同时确信编译程序知道这些变化，正是

这种能力使得内联汇编技艺超群。让我们进一步探索参数传递的奥秘吧。

2.4.1 输出操作数

第二行冒号后就是输出操作数，它是一个 C 表达式列表，其后的圆括号中是约束条件。对输出操作数而言，约束条件通常用“=”来修饰，表示“只写”的意思；修饰符“&”表示这是一个已被修改过的操作数，说明该指令使用这个操作数之前，它已经被修改过了。操作数之间要用逗号分开。

2.4.2 输入操作数

第三行的输入操作数遵循与输出操作数相同的语法，但无需只写修饰符“=”。

2.4.3 修改过的寄存器（已修改元素列表）

我们可以在汇编语句中修改各种各样的寄存器和内存，将其列出来，也可方便 gcc 知晓这些内容已被修改。

2.4.4 参数的编号方式

每个参数都给定一个位置编号，所有参数从 0 开始统一编号。例如，若有一个输出参数和两个输入参数，那么，%0 就是输出参数，%1 和 %2 都是输入参数。

2.4.5 约束条件

约束条件指明怎样使用操作数。GNU 文档中列出了所有简单约束和机器约束。表 2.4 列

举了 x86 最常用的约束。

表 2.4 x86 的简单约束和机器约束

约 束	作 用
a	寄存器 <code>eax</code>
b	寄存器 <code>ebx</code>
c	寄存器 <code>ecx</code>
d	寄存器 <code>edx</code>
S	寄存器 <code>esi</code>
D	寄存器 <code>edi</code>
I	常数 (0...31)
q	从 <code>eax</code> 、 <code>ebx</code> 、 <code>ecx</code> 、 <code>edx</code> 中动态分配一个寄存器
r	与 <code>q+esi</code> 、 <code>edi</code> 一样
m	内存定位
A	与 <code>a+b</code> 的作用相同。同时分配 <code>eax</code> 和 <code>ebx</code> ，形成一个 64 位寄存器

2.4.6 asm

实际上（尤其是在 Linux 内核中），由于关键字 `asm` 与其它结构同名，也许会在编译时引发错误。你常常看到写成“`_asm_`”的表达式，两者完全是一回事。

2.4.7 `_volatile_`

另一个常用的修饰词是 `_volatile_`，该修饰词对汇编代码的意义非同寻常，它告诉编译程序不要优化这个内联汇编程序。通常，随着硬件软件化，编译程序认为我们资源丰

富，浪费成性，于是尝试重写代码，使之尽可能高效。这对应用程序设计十分有用，但在硬件级程序设计中适得其反。

例如，假定我们正向一个映射内存的寄存器中写数据，该寄存器由 `reg` 变量来表示。

接下来，我们开始对 `reg` 进行轮询操作。编译程序仅把这当作同一存储单元连续读操作，并去掉了明显冗余的部分。应用 `_volatile_` 修饰符后，编译程序就知道不要优化使用了该变量的存取操作。同样，当你看到内联汇编代码块中出现 `asm volatile (...)` 字样时，编译程序也不会优化这一代码块。

现在，我们基本了解了汇编和 `gcc` 内联汇编，可以来看一看真正的内联汇编代码了。

我们首先运用所学知识来探究一个简单的例子，之后才是稍复杂一些的代码块。

下面是第一个例子的代码，它将变量传给一个内联汇编代码块：

```
6 int foo(void)
```

```
7 {
```

```
8 int ee = 0x4000, ce = 0x8000, reg;
```

```
9  __asm__ __volatile__("movl %1, %%eax":
10  "movl %2, %%ebx";
11  "call setbits" ;
12  "movl %%eax, %0"
13  : "=r" (reg) // reg [param %0] is output
14  : "r" (ce), "r"(ee) // ce [param %1], ee [param %2] are inputs
15  : "%eax" , "%ebx" // %eax and % ebx got clobbered
16  )
17  printf("reg=%x",reg);
18 }
```

6行:

本行是C例程的开始。

8行:

局部变量 `ee`、`ce`、`reg` 作为参数传给内联汇编程序。

9 行:

本行是内联汇编例程的开始，它将 `ce` 的值传给 `eax`。

10 行:

将 `ee` 的值传给 `ebx`。

11 行:

在汇编程序中调用函数。

12 行:

将返回值存储到 `eax` 中，并将其复制给 `reg`。

13 行:

本行是输出参数列表，参数 `reg` 的属性为只写。

14 行:

本行是输入参数列表，`ce` 和 `ee` 是寄存器变量。

15 行:

本行是寄存器修改列表。该例程改变了寄存器 `eax` 和 `ebx` 的值，编译程序知道在执行

这个例程后不要使用 `eax` 和 `ebx` 的值。

16 行:

本行标志着内联汇编例程的结束。

第二个例子是函数 `switch_to()` 的应用。该函数在头文件 `include/asm-i386/system.h`

中定义，是 Linux 上下文切换的核心部分。本章仅探讨它的内联汇编机制，第九章“编译

Linux 内核”将详细分析如何使用 `switch_to()`。

```
include/asm-i386/system.h
```

```
012 extern struct task_struct * FASTCALL(__switch_to(struct task_struct *prev,
```

```
struct
```

```
    task_struct *next));
```

```
...
```

```
015 #define switch_to(prev,next,last) do {
```

```
016     unsigned long esi,edi;
```

```
017  asm volatile("pushfl\n\t"

018  "pushl %%ebp\n\t"

019  "movl %%esp,%0\n\t" /* save ESP */

020  "movl %5,%%esp\n\t" /* restore ESP */

021  "movl $1f,%1\n\t" /* save EIP */

022  "pushl %6\n\t" /* restore EIP */

023  "jmp __switch_to\n"

023  "1:\t"

024  "popl %%ebp\n\t"

025  "popfl"

026  :="m" (prev->thread.esp),"=m" (prev->thread.eip),

027  "=a" (last),"=S" (esi),"=D" (edi)

028  : "m" (next->thread.esp),"m" (next->thread.eip),

029  "2" (prev), "d" (next));

030  } while (0)
```

12行:

FASTCALL 告诉编译程序使用 registers 传递参数, 而 “asm linkage” 标志则告诉编译程序使用 stack 传递参数。

15行:

对编译程序来说, do{statements...}while(0)这样的编码方式使得宏看起来更像一个函数, 因此允许使用局部变量。

16行:

不要弄错啦: 这只是局部变量的名字。

17行:

这就是内联汇编程序, 编译时不需要优化。

23行:

参数 1 被用作返回地址。

17-24行:

\n\t 与编译程序/汇编程序的接口有关。每条汇编指令都应该在各自的线路上运行。

26 行:

prev->thread.esp 和 prev->thread.eip 是输出参数:

[%0]= (prev->thread.esp), 是只写内存

[%1]= (prev->thread.eip),是只写内存

27 行:

[%2]=(last), 对寄存器 eax 只写

[%3]=(esi), 对寄存器 esi 只写

[%4]=(edi), 寄存器 edi 只写

28 行:

这是输入参数:

[%5]= (next->thread.esp), 是内存

[%6]= (next->thread.eip), 是内存

29 行:

[%7]= (prev), 重新使用 2 号参数 (寄存器 eax) 作为输入

[%8]= (next), 是赋给寄存器 edx 的一个输入

注意: 此处没有已修改寄存器列表。

PowerPC 中的内联汇编程序与 x86 中的结构几乎完全一样。“m”、“r”这样的简单约束条件与一组 PowerPC 的机器约束一起使用。以下是一个交换 32 位指针的例程, 注意看看其语法与 x86 的语法有多神似:

```
include/asm-ppc/system.h

103 static __inline__ unsigned long
104 xchg_u32(volatile void *p, unsigned long val)
105 {
106     unsigned long prev;
107
108     __asm__ __volatile__ ("l\n\
109     l: 1warx    %0,0,%2 \n"
```

110

111 " stwcx. %3,0,%2 \n\
"

112 bne- 1b"

113 : "&r" (prev), "=m" (*(volatile unsigned long *)p)

114 : "r" (p), "r" (val), "m" (*(volatile unsigned long *)p)

115 : "cc", "memory");

116

117 return prev;

118 }

103 行:

这一子例程将被适当地扩展，不会被调用。

104 行:

带参数 p 和 val 的例程名。

106 行:

局部变量 `prev`。

108 行:

内联汇编程序，编译时无需优化。

109—111 行:

`lwarx` 和 `stwcx` 形成“原子交换操作”。`lwarx` 从内存加载一个字并保存其地址，用于存储其后 `stwcx` 的结果。

112 行:

不相等时转到标号 1 处 (b: 向后跳转)。

113 行:

这是输出操作数:

`[%0]= (prev)`, 只写, 已修改

`[%1]= (*(volatile unsigned long *)p)`, 只写内存操作数

114 行:

这是输入操作数:

[%2]= (p), 寄存器操作数

[%3]= (val),寄存器操作数

[%4]= (*(volatile unsigned long *)p), 内存操作数

115 行:

这是被修改过的操作数:

[%5]= 条件代码寄存器被修改

[%6]=内存被修改

关于汇编语言以及 Linux2.6 内核如何应用汇编语言的讨论到此结束。我们先后分析了 PPC 和 x86 体系结构的汇编语言之间的差别，以及与平台无关时如何应用通用 ASM 程序设计技术。Linux 内核的大部分代码都是用 C 语言编写的，现在，我们该来关注 C 语言程序设计了，当然，还要探讨程序员们在使用 C 语言的过程中遇到的一些常见问题。

2.5 特殊的 C 语言用法

Linux 内核中的许多规范都需要经过反复查找和阅读才能发现其最终的意义和目的。

本节将着眼于贯穿整个 Linux2.6 内核中的常见 C 语言规范，澄清 C 语言用法中几个含糊

不清、容易误解的地方。

2.5.1 asmlinkage

`asmlinkage` 告诉编译程序要使用局部堆栈来传递参数，这就涉及到了宏 `FASTCALL`，

它通知（体系结构相关的）编译程序使用通用寄存器传递参数。以下是来自

`include/asm/linkage.h` 的宏：

`include/asm/linkage.h`

```
4 #define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
```

```
5 #define FASTCALL(x) x __attribute__((regparm(3)))
```

```
6 #define fastcall __attribute__((regparm(3)))
```

下面是 `asmlinkage` 的一个例子：

```
asmlinkage long sys_gettimeofday(struct timeval __user *tv, struct timezone
__user *tz)
```

2.5.2 UL

UL 常用在数值常数之后，标明该常数为“unsigned long”类型。UL（L 代表 long）负责告诉编译程序将这一数值当作 long 型数值来处理，因此，使用 UL（或是 L）很有必要，它能够保证特定体系结构内的数据不会溢出其数据类型所规定的范围。例如：一个 16 位整数的值位于 -32768 和 +32767 之间，一个无符号整数的值可以达到 65535。涉及到很大的数或长的位掩码时，使用 UL 有助于编写出体系结构无关的代码。

内核代码中有一些这样的例子，例如：

```
-----  
-----
```

```
include/linux/hash.h
```

```
18 #define GOLDEN_RATIO_PRIME 0x9e370001UL
```

```
include/linux/kernel.h
```

```
23 #define ULONG_MAX (~0UL)
```

```
include/linux/slab.h
```

```
39 #define SLAB_POISON    0x00000800UL /* Poison objects */
```

2.5.3 inline

关键字 `inline` 表明要优化函数的可执行代码，这可以通过将函数的代码合并到调用程序的代码中来实现。Linux 内核使用的 `inline` 函数大多被声明为 `static` 类型。一个“`static inline`”的函数促使编译程序尝试着将其代码插入到所有调用它的程序中。可能

的话，丢弃该函数的汇编代码。偶尔，编译程序也不能丢弃这些汇编代码（以免循环中还要使用），但就绝大多数情况而言，声明为 `static inline` 的函数意味着直接将它加入到调用程序中。

这一合并能够免除函数调用的任何开销，`#define` 语句也可以排除额外的函数调用，其典型应用参见嵌入式系统交叉编译程序的可移植性。

既然如此，为何不始终使用 `inline` 呢？原因在于使用 `inline` 会增加二进制映像的大小，而这可能会降低访问 CPU 高速缓存的速度。

2.5.4 `const` 和 `volatile`

对许多新手而言，这两个关键字简直是祸根。`const` 不一定是代表常数，有时它表示“只读”的意思。例如，“`const int *x`”中 `x` 是一个指向 `const` 整数的指针，因此，可以修改该指针，但不能修改这个整数；而在“`int const *x`”中，`x` 却是一个指向整数的 `const` 指针，因而这个整数可以改变，但指针 `x` 却不行。以下是一个有关 `const` 的例子：


```
include/asm-i386/processor.h

628 static inline void prefetch(const void *x)

629 {

630     __asm__ __volatile__ ("dcbt 0,%0" :: "r" (x));

631 }
```

关键字 `volatile` 表明变量无需警告就可以被修改。它通知编译程序每次使用该变量时都要重新加载其值，而不是存储并访问一个副本。中断处理、硬件寄存器，以及并发进程之间共享的变量都是被标记为 `volatile` 的典型例子。以下是如何使用 `volatile` 的一个例子：

```
include/linux/spinlock.h

51 typedef struct {
```

...

```
volatile unsigned int lock;
```

...

```
58 } spinlock_t;
```

假定 `const` 是“只读”的意思，那么，某些特定的变量就可以既是 `const` 类型又是 `volatile` 类型（比如说，一个保存某一只读硬件寄存器内容的变量，其值有规律地改变）。

该 C 语言用法掠影为 Linux 内核爱好者们展开了绚丽的画卷，希望能够引导他们在阅读内核源代码时回到正确的轨道上来。

2.6 内核探测工具一览

成功编译并构建自己的 Linux 内核后，你也许很想窥视内核在运行前、运行后，甚至运行过程中其内部的奥秘。本节大致介绍 Linux 内核中常用的探究各种内核文件的工具。

2.6.1 objdump/readelf

`objdump` 和 `readelf` 可分别用于显示目标文件（对 `objdump` 而言）和 ELF 文件（对 `readelf` 而言）的任何信息。我们可以借助于命令行参数使用命令来查看给定的目标文件的文件头、文件大小及结构。例如，以下是一个简单的 C 程序（`a.out`）的 ELF 文件头，所用的 `readelf` 标志为 `h`：

```
Lwp> readelf h a.out
```

```
ELF Header:
```

```
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
```

```
Class:          ELF32
```

```
Data:           2's complement, little endian
```

```
Version:        1 (current)
```

```
OS/ABI:         UNIX - System V
```

```
ABI Version:    0
```

```
Type:           EXEC (Executable file)
```

```
Machine:        Intel 80386
```

Version: 0x1

Entry point address: 0x8048310

Start of program headers: 52 (bytes into file)

Start of section headers: 10596 (bytes into file)

Flags: 0x0

Size of this header: 52 (bytes)

Size of program headers: 32 (bytes)

Number of program headers: 6

Size of section headers: 40 (bytes)

Number of section headers: 29

Section header string table index: 26

这是该程序使用 `readelf` 标志 1 时的文件头:

```
Lwp> readelf 1 a.out
```

Elf file type is EXEC (Executable file)

Entry point 0x8048310

There are 6 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x000c0	0x000c0	R E	0x4
INTERP	0x0000f4	0x080480f4	0x080480f4	0x00013	0x00013	R	0x1

[Requesting program interpreter: /lib/ld-linux.so.2]

LOAD	0x000000	0x08048000	0x08048000	0x00498	0x00498	R E	0x1000
LOAD	0x000498	0x08049498	0x08049498	0x00108	0x00120	RW	0x1000
DYNAMIC	0x0004ac	0x080494ac	0x080494ac	0x000c8	0x000c8	RW	0x4
NOTE	0x000108	0x08048108	0x08048108	0x00020	0x00020	R	0x4

Section to Segment mapping:

Segment Sections...

00

01 .interp

```
02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn
.rel

.plt .init .plt .text .fini .rodata

03 .data .eh_frame .dynamic .ctors .dtors .got .bss

04 .dynamic

05 .note.ABI-tag
```

2.6.2 hexdump

命令 `hexdump` 可以显示给定十六进制/ASCII/八进制文件的内容。（注意：在老版本的Linux中，也使用 `od` (`octal dump`)。现在，绝大多数系统用 `hexdump` 取代了 `od`）

例如，要查看一个ELF文件 `a.out` 前64字节的十六进制表示的话，可以输入如下命令：

令：

```
Lwp> hexdump x n 64 a.out
```

```
00000000 457f 464c 0101 0001 0000 0000 0000 0000
```

```
00000010 0002 0003 0001 0000 8310 0804 0034 0000
```

```
0000020 2964 0000 0000 0000 0034 0020 0006 0028
```

```
0000030 001d 001a 0006 0000 0034 0000 8034 0804
```

```
0000040
```

值得注意的是（字节交换的）ELF 文件头在地址 0x0000000 处的魔数。

它在调试时特别有用：当硬件设备将其状态转储到某个文件时，常规的文本编辑器一般将它解释成包含很多控制字符的文件，`hexdump` 命令使得你无需干预编辑器的转换工作就可以看到文件中究竟包含了些什么。编辑器 `hexedit` 能帮助你直接修改该文件，而不必先将其内容转换成 ASCII 码（或 Unicode 编码）。

2.6.3 nm

实用程序 `nm` 可以列出指定目标文件中的符号，它能够显示符号的值、类型和名字，虽然不如其他程序一样有用，但调试库文件时，它就能大显身手了。

2.6.4 objcopy

当你想要复制一个目标文件而忽略或改变其某方面的内容时，可以使用 `objcopy` 命令。`objcopy` 的常见用法是去掉测试后正在运行的目标文件中的调试符号，这样做可以大

大减小目标文件的大小，因而常用于嵌入式系统中。

2.6.5 ar

ar（或 archive）命令有助于维护链接程序使用的索引函数库。ar 命令可以将一个或多个目标文件链接到一个程序库中，也可以从单个程序库中将目标文件分离出来。ar 命令最常用于 Make 文件，将一些常用的函数连接到单个库文件中。例如：你也许有这样一个例程：它可以分析一个命令文件并抽取特定的数据；或者有这样一个调用，它可以从硬件特定的寄存器内抽取信息。可能有多个可执行程序要使用这些例程，将它们归于单个库文件中有利于通过中央位置更好地进行版本控制。

2.7 内核发言：倾听来自内核的消息

当你的 Linux 系统更新后正在运行时，内核本身会记录一些消息，并提供整个操作过程中系统的状态信息。本节介绍几种最常见的 Linux 内核对终端用户“说话”的方式。

2.7.1 printk ()

最基本的内核通讯系统之一是 printk () 函数。内核使用 printk () 而不是 printf () 是因为内核没有链接标准 C 函数库，其实 printk () 的接口与 printf () 完

全一样，它可以在控制台显示多达 1024 个字符。`printk ()` 函数工作时首先设法获取控制台信号量，然后将要输出的字符存储到控制台的日志缓冲区，再调用控制台驱动程序来刷新缓冲区。若 `printk ()` 无法获得控制台信号量，就只能把要输出的字符存储到日志缓冲区，并依赖拥有控制台信号量的进程来刷新这个缓冲区。在 `printk ()` 存储任何数据到日志缓冲区之前，必须使用日志缓冲区锁，这样才能保证并发调用 `printk ()` 时不会出错。如果已经获得了控制台信号量，那么刷新日志缓冲区之前，可以多次调用 `printk ()`。所以，别用 `printk ()` 语句来标明任何程序的测试时间。

2.7.2 dmesg

Linux 内核有多种方式可用于存储日志和信息。`sysklogd ()` 是 `syslogd ()` 和 `klogd ()` 的组合（详情可参阅这些命令的相关手册，此处仅作简单概述）。Linux 内核通过 `klogd ()` 发送消息，并给它标上适当的警告级。所有级别的消息都存储在 `/proc/kmsg` 中。`dmesg` 是一个命令行工具，可用于显示存储在 `/proc/kmsg` 中的缓冲区内容，并能够根据消息级别来选择是否要过滤这个缓冲区。

2.7.3 /var/log/messages

Linux 系统的 `/var/log/messages` 下存储的是大多数已登录系统的信息。对于某些特定的单元，存储了已接收的消息，可以借助 `syslogd` () 程序来读取 `/etc/syslogd.conf` 的内容。日志信息可借助 `syslogd.conf` 中的条目存储到许多文件中去，但该文件在不同版本的 Linux 系统中有所不同，不过通常是放在 `/var/log/messages` 下。

2.8 其他

当内核设计者们开始漫步在内核代码中时，那些曾经困扰过他们的问题简直包罗万象，本节将这些内容概括于此，旨在鼓励你探索 Linux 内部的奥秘。

2.8.1 `_init`

宏 `_init` 告知编译程序相关的函数和变量仅用于初始化。编译程序将标有 `_init` 的所有代码存储到特殊的内存段中，初始化结束后就释放这段内存：

`drivers/char/random.c`

```
679 static int __init batch_entropy_init(int size, struct entropy_store *r)
```

例如，随机设备驱动程序在被加载前初始化一个熵池，加载该驱动程序时，可以使用不同的函数来加大或减小熵池。这个初始化过程就算不是一个标准过程，至少也是很普遍的。

与之类似，如果某些数据只在初始化时才用到，这些数据就必需标记为 `_initdata`。

我们来看看在 ESP 设备驱动程序中如何使用 `_initdata`：

`drivers/char/esp.c`

```
107 static char serial_name[] __initdata = "ESP serial driver";
```

```
108 static char serial_version[] __initdata = "2.2";
```

同样，宏 `_exit` 和 `_exitdata` 仅用于退出和关闭例程，一般在注销设备驱动程序时才

使用。

2.8.2 likely () 和 unlikely ()

Linux 内核开发者用宏 `likely ()` 和 `unlikely ()` 向编译程序和芯片集给予提示。现代 CPU 具有精确的启发式分支预测法，它尝试着预测下一条到来的命令，以便达到最快的速度。宏 `likely ()` 和 `unlikely ()` 允许开发者通过编译程序告诉 CPU：某一代码段很可能被执行，因而应该预测到；某一代码段很可能不被执行，不必预测。

如果对指令流水线技术有一定理解的话，就能明白分支预测的重要性了。现代处理器可以预取指令，就是说，它们预测接下来会被执行的几条指令，并把这些指令加载到处理器中，检测后被分派到处理器的各种单元（例如整数、浮点数等），以期得到最好的执行效果。处理器可能会推迟某些指令的执行，以便等待前一指令执行后产生的中间结果。现在，想象有一个指令流，处理器加载了其中一个分支指令，使得它有两个继续预取指令的指令流，若处理器常常无法作出聪明的选择，它将花费太多时间来重新加载需要执行的指令。如果处理器得到分支程序可能会怎么走的暗示呢？在某些体系结构中，分支预测的一种简单方法就是考查这个分支的目标地址，若其值在当前地址之前，那么，这一

分支很可能处于一个循环结构的末端，从这里返回循环执行多次，只有一次失败，从而退出循环。

借助于特殊助记手段，可以利用软件来克服体系结构相关的分支预测。编译程序通过函数 `_builtin_expect ()` 来实现这一功能，而 `_builtin_expect ()` 正是宏 `likely ()` 和 `unlikely ()` 的基础。

如上所述，分支预测和处理器流水线技术都是错综复杂的，也不在本书的讨论之列。

但是，“调整”那些我们认为很重要的代码总可以提高性能。考虑如下代码块：

kernel/time.c

```
90 asmlinkage long sys_gettimeofday(struct timeval __user *tv, struct timezone
__user *tz)

91 {

92     if (likely(tv != NULL)) {

93         struct timeval ktv;
```

```
94     do_gettimeofday(&ktv);

95     if (copy_to_user(tv, &ktv, sizeof(ktv)))

96         return -EFAULT;

97 }

98 if (unlikely(tz != NULL)) {

99     if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))

100         return -EFAULT;

101 }

102 return 0;

103 }
```

在这些代码中，我们发现用于获得时间的系统调用可能有一个非空的 `timeval` 结构（见 92-96 行）。若它为 `NULL`，我们就无法填入所请求的时间啦！时区也未必非空（见 98-100 行）。换言之，调用者通常会查询时间而往往极少询问时区。

`likely ()` 和 `unlikely ()` 的特殊实现定义如下⁴:

```
-----  
-----  
  
include/linux/compiler.h  
  
45 #define likely(x) __builtin_expect(!!(x), 1)  
  
46 #define unlikely(x) __builtin_expect(!!(x), 0)  
  
-----  
-----
```

2.8.3 IS_ERR 和 PTR_ERR

宏 `IS_ERR` 将负的错误号编码成指针，而宏 `PTR_ERR` 则将该指针恢复成错误号。

这两个宏均在 `include/linux/err.h` 中定义。

2.8.4 通告程序链 (Notifier Chains)

发生可变异步事件时，通告程序链机制为内核提供它感兴趣的通告信息。这个通用接

⁴ 代码引用中的 `__builtin_expect()`，在 GCC2.96 之前一直是无效的，因为在此之前的 GCC 无法影响分支预测。

口将其可用性扩展到了内核的所有子系统和组件中。

通告程序链 (`notifier chain`) 就是一个 `notifier_block` 对象的单链表:

```
-----  
-----  
  
include/linux/notifier.h  
  
14 struct notifier_block  
  
15 {  
  
16 int(*notifier_call)(struct notifier_block *self, unsigned long, void *);  
  
17 struct notifier_block *next;  
  
18 int priority;  
  
19 };  
  
-----  
-----
```

`notifier_block` 包含指向函数 `notifier_call` 的指针, 当事件发生时调用该函数, 其参数包括指向保存信息的 `notifier_block` 对象的指针、相应事件代码或标志的值, 以及

指向特定子系统数据类型的指针。

`notifier_block` 结构还包含指向链中下一个 `notifier_block` 的指针和优先级声明。

例程 `notifier_block_register ()` 和 `notifier_block_unregister ()` 分别用于向特

定通告程序链注册或注销一个 `notifier_block` 对象。

小结:

本章阐述了许多探索 Linux 内核之前应当了解的背景知识，介绍了两种动态存储方法——链表和二叉搜索树；对这些数据结构的基本了解有助于你在其他章节中讨论进程和分页机制；接下来介绍了汇编语言的基础知识，以便在机器层进行探索和调试。同时，关于内联汇编程序，我们展示了同一函数中出现 C 代码和汇编代码的情况。最后，本章讨论了研究内核的各个方面的必需的各种命令和函数。

项目: Hellomode

本节介绍了一些基本概念，这是理解稍后将讨论的其他 Linux 概念和数据结构必备的知识。该项目重点在于使用新的 2.6 驱动程序结构来创建一个可加载模块，并为后面的项目编译该模块。提到设备驱动程序，问题马上就变得复杂了，因此，我们只介绍 Linux

模块的基本结构，在后面的项目中再来介绍这个驱动程序。该模块在 PPC 上和 x86 上均可运行。

第一步：构造 Linux 模块的框架

我们写的第一个模块是基本的“hello world”字符设备驱动程序。首先，考虑该模块的基本代码，然后示范怎样使用新的 2.6 Makefile 系统（详情参见第九章），最后，分别使用 `insmod` 命令和 `rmod` 命令加载或移除⁵该模块。

```
-----  
  
-----  
  
hellomod.c  
  
001  
  
// hello world driver for Linux 2.6  
  
004 #include <linux/module.h>  
  
005 #include <linux/kernel.h>
```

⁵ 要确定在配置内核时允许卸载模块。

```
006 #include <linux/init.h>

007 #MODULE_LICENCE("GPL"); //get rid of taint message

009 static int __init lkp_init( void )

{

    printk("<l>Hello,World! from the kernel space...\n");

    return 0;

013 }

015 static void __exit lkp_cleanup( void )

{

    printk("<l>Goodbye, World! leaving kernel space...\n");

018 }

020 module_init(lkp_init);

021 module_exit(lkp_cleanup);
```

4 行:

所有模块都要使用头文件 `module.h`，此文件必须包含进来。

5 行:

头文件 `kernel.h` 包含了常用的内核函数。

6 行:

头文件 `init.h` 包含了宏 `_init` 和 `_exit`，它们允许释放内核占用的内存。建议浏览一下该文件中的代码和注释。

7 行:

提示可能没有 GNU 公共许可证。有几个宏是在 2.4 版的内核中才开发的（详情参见 `modules.h`）。

9-12 行:

这是模块的初始化函数，它必需包含诸如要编译的代码、初始化数据结构等内容。11

行用 `printk ()` 从内核发送消息，并提示加载模块后从何处读取该消息。

15-18 行:

这是模块的退出和清理函数。此处可以做所有终止该驱动程序时相关的清理工作。

20 行:

这是驱动程序初始化的入口点。对于内置模块，内核在引导时调用该入口点；对于可加载模块则在该模块插入内核时才调用。

21 行:

对于可加载模块，内核在此处调用 `cleanup_module ()` 函数，而对于内置的模块，它什么都不做。

在该驱动程序中，仅有一个初始化 (`module_init`) 点和一个清理 (`cleanup_exit`) 点。加载或卸载模块时，内核会来寻找这些函数。

第二步：编译模块

如果你习惯使用老办法来编译内核模块（例如，从 `#define MODULE` 开始），就会发现新的方法有很大变化。即使是首次编译 2.6 的模块，看起来也相当简单。该模块的

`Makefile` 文件基本内容如下:

Makefile

```
002 # Makefile for Linux Kernel Primer module skeleton (2.6.7)
```

```
006 obj-m += hellomod.o
```

要注意的是，需要向编译系统特别声明该模块要编译成可加载模块。该 Makefile 文

件的命令行调用由称为 `doit` 的脚本文件来打包，如下所示：

```
-----doit
```

```
001 make -C /usr/src/linux-2.6.7 SUBDIRS=$PWD modules
```

```
-----
```

1行：

C 选项告诉 `make` 程序读取 Makefiles 或做其他任何事之前，先要修改 Linux 源目录

(本例中是 `/usr/src/linux-2.6.7`) 。

执行./doit 后可得到与以下内容类似的输出结果:

```
Lkp# ./doit
```

```
make: Entering directory '/usr/src/linux-2.6.7'
```

```
CC [M] /mysource/hellomod.o
```

```
Building modules, stage 2
```

```
MODPOST
```

```
CC /mysource/hellomod.o
```

```
LD [M] /mysource/hellomod.ko
```

```
make: Leaving directory '/usr/src/linux-2.6.7'
```

```
lkp# _
```

如果在 Linux 早期的版本上编译过或创建过 Linux 模块，那么此处还有一个链接步

骤 LD，其输出模块是 hellomod.ko。

第三步：运行代码

现在我们已经准备好，可以将新的模块插入到内核中啦！这可以用命令 insmod 来实

现，如下所示：

```
1kp# insmod hellomod.ko
```

`lsmod` 命令可用于检查模块是否正确插入到内核中了：

```
1kp# lsmod
```

```
Module      Size  Used by
```

```
hellomod    2696  0
```

```
1kp#
```

模块的输出由 `printk ()` 来产生。该函数默认打印系统文件 `/var/log/messages` 的内

容。快速浏览这些消息可输入如下命令：

```
1kp# tail /var/log/messages
```

这一命令打印日志文件的最后 10 行内容，可以看到我们的初始化信息：

```
...
```

```
...
```

```
Mar 6 10:35:55 1kpl kernel: Hello,World! from the kernel space...
```

使用 `rmmod` 命令，加上我们在 `insmod` 中看到的模块名，可以从内核中移除该模块

(还可以看到退出时显示的信息)。如下所示：

```
1kp# rmmod hellomod
```

同样，输出的内容也在日志文件中，如下所示：

```
...
```

```
...
```

```
Mar 6 12:00:05 1kpl kernel: Hello,World! from the kernel space...
```

根据 x-系统的配置或者是否有基本命令行，`printk` 的输出可以在终端上显示，也可

以存放在日志文件中。在下一个项目中，考虑系统的任务变量时会再提到这个问题。

习题：

1、描述哈希表在 Linux 内核中如何实现？

2、双向链表中的元素有一个 `list_head` 结构，内核采用该结构之前，它已有指向其他相

似结构的 `prev` 指针和 `next` 指针。创建一个仅有 `prev` 和 `next` 指针的结构其目的是什么？

3、什么是内联汇编？为什么要使用它？

4、假定要写一个访问串口寄存器的设备驱动程序。你会将这些地址标记为 `volatile` 吗？

为什么？

5、想象一下 `_init` 做些什么，你认为什么类型的函数会使用这个宏？