**Linux System.map文件**

2016年04月24日 14:52:36 [小近视爱老花镜](https://me.csdn.net/YSBJ123) 阅读数：1974 标签： [System.map](http://so.csdn.net/so/search/s.do?q=System.map&t=blog" \t "_blank) 更多

个人分类： [Linux](https://blog.csdn.net/YSBJ123/article/category/5708269)

当运行GNU链接器gld（ld）时若使用了"-M"选项，或者使用nm命令，则会在标准输出设备（通常是屏幕）上打印出链接映像（link map）信息，即是指由链接程序产生的目标程序内存地址映像信息。其中列出了程序段装入到内存中的位置信息。具体来讲有如下信息：

目标文件及符号信息映射到内存中的位置。

公共符号如何放置。

链接中包含的所有文件成员及其引用的符号。

通常我们会把发送到标准输出设备的链接映像信息重定向到一个文件中（例如System.map）。在编译内核时，linux/Makefile文件产生的System.map文件就用于存放内核符号表信息。符号表是所有内核符号及其对应地址的一个列表，当然也包括上面说明的\_etext、\_edata和\_end等符号的地址信息。随着每次内核的编译，就会产生一个新的对应System.map文件。当内核运行出错时，通过System.map文件中的符号表解析，就可以查到一个地址值对应的变量名，或反之。

利用System.map符号表文件，在内核或相关程序出错时，就可以获得我们比较容易识别的信息。符号表的样例如下所示：

|  |
| --- |
| c03441a0 B dmi\_broken  c03441a4 B is\_sony\_vaio\_laptop  c03441c0 b dmi\_ident  c0344200 b pci\_bios\_present  c0344204 b pirq\_table |

其中每行说明一个符号，第1栏指明符号值（地址）；第2栏是符号类型，指明符号位于目标文件的哪个区（sections）或其属性；第3栏是对应的符号名称。

第2栏中的符号类型指示符通常有表3-5所示的几种，另外还有一些与采用的目标文件格式相关。如果符号类型是小写字符，则说明符号是局部的；如果是大写字符，则说明符号是全局的（外部的）。参见文件include/a.out.h中nlist{}结构n\_type字段的定义（第110～185行）。

表3-5  目标文件符号列表文件中的符号类型

|  |  |  |
| --- | --- | --- |
| 符符号类型 | 名称 | 说明 |
| A | Absolute | 符号的值是绝对值，并且在进一步链接过程中不会被改变 |
| B | BSS | 符号在未初始化数据区或区（section）中，即在BSS段中 |
| C | Common | 符号是公共的。公共符号是未初始化的数据。在链接时，多个公共符号可能具有同一名称。如果该符号定义在其他地方，则公共符号被看作是未定义的引用 |
| D | Data | 符号在已初始化数据区中 |
| G | Global | 符号是在小对象已初始化数据区中的符号。某些目标文件的格式允许对小数据对象（例如一个全局整型变量）可进行更有效的访问 |
| I | Inderect | 符号是对另一个符号的间接引用 |
| N | Debugging | 符号是一个调试符号 |
| R | Read only | 符号在一个只读数据区中 |
| S | Small | 符号是小对象未初始化数据区中的符号 |
| T | Text | 符号是代码区中的符号 |
| U | Undefined | 符号是外部的，并且其值为0（未定义） |
| - | Stabs | 符号是a.out目标文件中的一个stab符号，用于保存调试信息 |
| ? | Unknown | 符号的类型未知，或者与具体文件格式有关 |

可以看出名称为dmi\_broken的变量位于内核地址0xc03441a0处。

System.map位于使用它的软件（例如内核日志记录后台程序klogd）能够寻找到的地方。在系统启动时，如果没有以一个参数的形式为klogd给出System.map的位置，则klogd将会在三个地方搜寻System.map。依次为：

/boot/System.map

/System.map

/usr/src/linux/System.map

尽管内核本身实际上不使用System.map，但其他程序，像klogd、lsof、ps以及dosemu等许多软件都需要有一个正确的System.map文件。利用该文件，这些程序就可以根据已知的内存地址查找出对应的内核变量名称，便于对内核的调试工作。

**System.map解析**

第一部分：System.map的作用

有关System.map文件的信息好象很缺乏。其实它一点也不神秘，并且在整个事情当中它并不象看上去那么得重要。但是由于缺乏必要的文档说明，使其显得比较神秘。它就象耳垂，我们每个人都有，但却不知道是干什么用的。注意，我并不会是百分之一百正确的。例如，一个系统很可能没有/proc文件系统支持，但是大多数系统肯定有。这里我假定你是“随大流的”，并有一个典型配置的系统。  
某些有关内核出错(oops)的阐述来自于Alessandro Rubini的“Linux设备驱动程序” 一书，我是从其中学到大部分内核编程知识的。  
**什么是符号(Symbols)?**    在编程中，一个符号(symbol)是一个程序的创建块：它是一个变量名或一个函数名。 正如你自己编制的程序一样，内核具有各种符号也是不应该感到惊奇的。当然，区别在于内核是一非常复杂的代码块，并且含有许多、许多的全局符号。  
内核符号表(Kernel Symbol Table)是什么东西?  
    内核并不使用符号名。它是通过变量或函数的地址(指针)来使用变量或函数的，而不是使用size\_t BytesRead，内核更喜欢使用(例如)c0343f20来引用这个变量。  
而另一方面，人们并不喜欢象c0343f20这样的名字。我们跟喜欢使用象size\_t BytesRead这样的表示。通常，这并不会带来什么问题。内核主要是用C语言写成的，所以在我们编程时编译器/连接程序允许我们使用符号名，并且使内核在运行时使用地址表示。这样大家都满意了。  
    然而，存在一种情况，此时我们需要知道一个符号的地址（或者一个地址对应的符号）。这是通过符号表来做到的，与gdb能够从一个地址给出函数名（或者给出一个函数名的地址）的情况很相似。符号表是所有符号及其对应地址的一个列表。这里是一个符号表例子：  
   c03441a0 B dmi\_broken  
   c03441a4 B is\_sony\_vaio\_laptop  
   c03441c0 b dmi\_ident  
   c0344200 b pci\_bios\_present  
   c0344204 b pirq\_table  
   c0344208 b pirq\_router  
   c034420c b pirq\_router\_dev  
   c0344220 b ascii\_buffer  
   c0344224 b ascii\_buf\_bytes  
    你可以看出名称为dmi\_broken的变量位于内核地址c03441a0处。   
**什么是System.map文件?**    有两个文件是用作符号表的:  
       1. /proc/kallsyms  
       2. System.map   
    这里，你现在可以知道System.map文件是干什么用的了。  
    每当你编译一个新内核时，各种符号名的地址定会变化。  
/proc/ksyms 是一个 "proc文件" 并且是在内核启动时创建的。实际上它不是一个真实的文件；它只是内核数据的简单表示形式，呈现出象一个磁盘文件似的。如果你不相信我，那么就试试找出/proc/ksyms的文件大小来。因此， 对于当前运行的内核来说，它总是正确的  
    然而，System.map却是文件系统上的一个真实文件。当你编译一个新内核时，你原来的System.map中的符号信息就不正确了。随着每次内核的编译，就会产生一个新的System.map文件，并且需要用该文件取代原来的文件。  
**什么是一个Oops?**    在自己编制的程序中最常见的出错情况是什么?是段出错(segfault)，信号11。  
Linux内核中最常见的bug是什么?也是段出错。除此，正如你想象的那样，段出错的问题是非常复杂的，而且也是非常严重的。当内核引用了一个无效指针时，并不称其为段出错 --而被称为"oops"。一个oops表明内核存在一个bug，应该总是提出报告并修正该bug。  
    请注意，一个oops与一个段出错并不是一回事。你的程序并不能从段出错中恢复过来，当出现一个oops时，并不意味着内核肯定处于不稳定的状态。Linux内核是非常健壮的；一个oops可能仅杀死了当前进程，并使余下的内核处于一个良好的、稳定的状态。  
一个oops并非是内核死循环(panic)。在内核调用了panic()函数后，内核就不能继续运行了；此时系统就处于停顿状态并且必须重启。如果系统中关键部分遭到破坏,那么一个oops也可能会导致内核进入死循环(panic)。例如，设备驱动程序中出现的oops就几乎不会导致系统进行死循环。  
    当出现一个oops时，系统就会显示出用于调试问题的相关信息，比如所有CPU寄存器中的内容以及页描述符表的位置等，尤其会象下面那样打印出EIP(指令指针)的内容：  
   EIP: 0010:[]  
   Call Trace: []  
          
**一个Oops与System.map文件有什么关系呢?**    我想你也会认为EIP和Call Trace所给出的信息并不多，但是重要的是，对于内核开发人员来说这些信息也是不够的。由于一个符号并没有固定的地址， c010b860可以指向任何地方。  
为了帮助我们使用oops含糊的输出，Linux使用了一个称为klogd(内核日志后台程序)的后台程序，klogd会截取内核oops并且使用syslogd将其记录下来，并将某些象c010b860的信息转换成我们可以识别和使用的信息。换句话说，klogd是一个内核消息记录器(logger)，它可以进行名字-地址之间的解析。一旦klogd开始转换内核消息，它就使用手头的记录器，将整个系统的消息记录下来，通常是使用syslogd记录器。

    为了进行名字-地址解析，klogd就要用到System.map文件。我想你现在知道一个oops与System.map的关系了。  
    深入说明： 其实klogd会执行两类地址解析活动。  
        \* 静态转换，将使用System.map文件。  
        \* 动态转换，该方式用于可加载模块，不使用System.map，因此与本讨论没有关系，但我仍然对其加以简单说明。  
        Klogd动态转换  
假设你加载了一个产生oops的内核模块。于是就会产生一个oops消息，klogd就会截获它，并发现该oops发生在d00cf810处。由于该地址属于动态加载模块，因此在System.map文件中没有对应条目。klogd将会在其中寻找并会毫无所获，于是断定是一个可加载模块产生了oops。此时klogd就会向内核查询该可加载模块输出的符号。即使该模块的编制者没有输出其符号，klogd也起码会知道是哪个模块产生了oops，这总比对一个oops一无所知要好。   
    还有其它的软件会使用System.map，我将在后面作一说明。  
System.map应该位于什么地方?  
    System.map应该位于使用它的软件能够寻找到的地方，也就是说，klogd会在什么地方寻找它。在系统启动时，如果没有以一个参数的形式为klogd给出System.map的位置，则klogd将会在三个地方搜寻System.map。依次为：  
       1. /boot/System.map  
       2. /System.map  
       3. /usr/src/linux/System.map   
System.map  
同样也含有版本信息，并且klogd能够智能化地搜索正确的map文件。例如，假设你正在运行内核2.4.18并且相应的map文件位于  
/boot/System.map。现在你在目录/usr/src/linux中编译一个新内核2.5.1。在编译期间，文件/usr/src/linux/System.map就会被创建。当你启动该新内核时，klogd将首先查询/boot/System.map，确认它不是启动内核正确的map文件，就会查询 /usr/src/linux/System.map,确定该文件是启动内核正确的map文件并开始读取其中的符号信息。  
    几个注意点:  
\* 在2.5.x系列内核的某个版本，Linux内核会开始untar成linux-version，而非只是linux (请举手表决--有多少人一直等待着这样做?)。我不知道klogd是否已经修改为在/usr/src/linux-version/System.map中搜索。  
TODO：查看klogd源代码。  
        \* 在线手册上对此也没有完整描述，请看：  
   # strace -f /sbin/klogd | grep 'System.map'  
   31208 open("/boot/System.map-2.4.18", O\_RDONLY|O\_LARGEFILE) = 2  
                  
          显然，不仅klogd在三个搜索目录中寻找正确版本的map文件，klogd也同样知道寻找名字为 "System.map" 后加"-内核版本"，象 System.map-2.4.18. 这是klogd未公开的特性。  
有一些驱动程序将使用System.map来解析符号(因为它们与内核头连接而非glibc库等)，如果没有System.map文件，它们将不能正确地工作。这与一个模块由于内核版本不匹配而没有得到加载是两码事。模块加载是与内核版本有关，而与即使是同一版本内核其符号表也会变化的编译后内核无关。  
**还有谁使用了System.map?**    不要认为System.map文件仅对内核oops有用。尽管内核本身实际上不使用System.map，其它程序，象klogd，lsof，  
   satan# strace lsof 2>&1 1> /dev/null | grep System  
   readlink("/proc/22711/fd/4", "/boot/System.map-2.4.18", 4095) = 23  
          
    ps,  
   satan# strace ps 2>&1 1> /dev/null | grep System  
   open("/boot/System.map-2.4.18", O\_RDONLY|O\_NONBLOCK|O\_NOCTTY) = 6  
          
    以及其它许多软件，象dosemu，需要有一个正确的System.map文件。   
**如果我没有一个好的System.map，会发生什么问题?**    假设你在同一台机器上有多个内核。则每个内核都需要一个独立的 System.map文件！如果所启动的内核没有对应的System.map文件，那么你将定期地看到这样一条信息：  
        System.map does not match actual kernel (System.map与实际内核不匹配)  
    不是一个致命错误，但是每当你执行ps ax时都会恼人地出现。有些软件，比如dosemu，可能不会正常工作。最后，当出现一个内核oops时，klogd或ksymoops的输出可能会不可靠。  
我如何对上述情况进行补救?  
    方法是将你所有的System.map文件放在目录/boot下，并使用内核版本号重新对它们进行命名。假设你有以下多个内核：  
        \* /boot/vmlinuz-2.2.14  
        \* /boot/vmlinuz-2.2.13   
    那么，只需对应各内核版本对map文件进行改名，并放在/boot下，如：  
     
/boot/System.map-2.2.14  
   /boot/System.map-2.2.13  
**如果你有同一个内核的两个拷贝怎么办？**例如：  
        \* /boot/vmlinuz-2.2.14  
        \* /boot/vmlinuz-2.2.14.nosound   
    最佳解决方案将是所有软件能够查找下列文件：  
   /boot/System.map-2.2.14  
   /boot/System.map-2.2.14.nosound  
但是说实在的，我并不知道这是否是最佳情况。我曾经见到搜寻"System.map-kernelversion"，但是对于搜索"System.map  
-kernelversion.othertext"的情况呢?  
我不太清楚。此时我所能做的就是利用这样一个事实：/usr/src/linux是标准map文件的搜索路径，所以你的map文件将放在：  
        \* /boot/System.map-2.2.14  
        \* /usr/src/linux/System.map (对于nosound版本)   
    你也可以使用符号连接：  
   System.map-2.2.14  
   System.map-2.2.14.sound  
   System.map -> System.map-2.2.14.sound

【转：[http://edu.codepub.com/2011/0323/30291.php](http://edu.codepub.com/2011/0323/30291.php" \t "_blank)】

第二部分：System.map的格式

本文参考 binutils文档。

system.map内容格式为：线性地址 类型 符号

具体内容如下：   
00100000 A phys\_startup\_32  
c0100000 T startup\_32  
c0100000 A \_text                             注:表示内核代码第一个字节的地址  
c01000c6 t checkCPUtype  
c0100147 t is486  
c010014e t is386  
c0100199 t L6  
c010019b t check\_x87  
c01001c2 t setup\_idt  
c01001df t rp\_sidt  
c01001ec t ignore\_int  
c0100220 t rest\_init  
c0100220 T stext  
c0100220 T \_stext  
c0100252 t do\_pre\_smp\_initcalls  
c0100257 t run\_init\_process  
c0100283 t init  
c01003a8 t try\_name  
c0100529 T name\_to\_dev\_t  
c0100790 t calibrate\_delay\_direct

...

c02f95d7 T register\_kretprobe  
c02f96cd T unregister\_kretprobe  
c02f9760 t .text.lock.kprobes  
c02f97b0 T \_\_kprobes\_text\_end  
c02f9abe t iret\_exc  
c02fa1af A \_etext                              注:内核代码结束的位置，之后为内核初始化的数据  
c02fa1b0 A \_\_start\_\_\_ex\_table  
c02facc0 A \_\_stop\_\_\_ex\_table  
c02fb000 r \_\_func\_\_.12  
c02fb000 A \_\_start\_rodata  
c02fb00c r \_\_func\_\_.13  
c02fb020 r \_\_func\_\_.2  
c02fb02c r \_\_func\_\_.3  
c02fb040 R linux\_banner  
c02fb0bb r \_\_func\_\_.0  
c02fb0c7 r \_\_func\_\_.1  
c02fb0e0 r \_\_func\_\_.0  
c02fb0ec r \_\_func\_\_.1  
c02fb100 r border  
c02fb160 r cplens

...

c03e1b08 D ip\_statistics  
c03e1b10 D tcp\_statistics  
c03e1b18 D udp\_statistics  
c03e1b20 D icmp\_statistics  
c03e1b28 D net\_statistics  
c03e1b30 d fn\_hash\_kmem  
c03e1b34 d fn\_alias\_kmem  
c03e1b38 d xfrm\_dst\_cache  
c03e1b3c d secpath\_cachep  
c03e1b40 A \_edata                       注:内核初始化数据结束，之后为未初始化数据  
c03e2000 D init\_thread\_union  
c03e4000 A \_\_init\_begin  
c03e4000 t no\_halt  
c03e4000 T \_sinittext  
c03e400d t mca\_pentium  
c03e401d t no\_387  
c03e4033 t check\_fpu  
...

c043a058 b pfkey\_table  
c043a05c b pfkey\_table\_lock  
c043a05c b pfkey\_table\_users  
c043a060 b pfkey\_socks\_nr  
c043a064 b acqseq.4  
c043a068 b acqseq\_lock.5  
c043a068 A \_\_bss\_stop  
c043a068 A \_end                         注:内核未初始化数据结束  
c043b000 A pg0  
ffffe400 A \_\_kernel\_vsyscall  
ffffe410 A SYSENTER\_RETURN  
ffffe420 A \_\_kernel\_sigreturn  
ffffe440 A \_\_kernel\_rt\_sigreturn

符号类型.

小写字母表示局部; 大写字母表示全局(外部).   
A   
The symbol's value is absolute, and will not be changed by further linking.

B   
The symbol is in the uninitialized data section (known as BSS).

C   
The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references. For more details on common symbols, see the discussion of -warn-common in Linker options.

D   
The symbol is in the initialized data section.

G   
The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.

I   
The symbol is an indirect reference to another symbol. This is a GNU extension to the a.out object file format which is rarely used.

N   
The symbol is a debugging symbol.

R   
The symbol is in a read only data section.

S   
The symbol is in an uninitialized data section for small objects.

T   
The symbol is in the text (code) section.

U   
The symbol is undefined.

V   
The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

W   
The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

-   
The symbol is a stabs symbol in an a.out object file. In this case, the next values printed are the stabs other field, the stabs desc field, and the stab type. Stabs symbols are used to hold debugging information. For more information, see Stabs.

?   
The symbol type is unknown, or object file format specific. 

# info binutils查看更详细的信息

【转：<http://blog.csdn.net/gates84/archive/2006/11/27/1416514.aspx>】

看了一下Linux下的/boot/System.map，发现里边定义的东西还是比较有意思的  
仔细的看了一下，写一些东西：）  
  
部分摘录：  
    00000000 A agp\_frontend\_cleanup  
    c0127350 t kmem\_slab\_destroy  
    c0127430 T kmem\_cache\_create  
    c02a0ea0 R ide\_pio\_timings  
    c02aee28 r usb\_bandwidth\_option  
    c02c68bf ? \_\_kstrtab\_get\_zeroed\_page  
    c02c68cf ? \_\_kstrtab\_\_\_free\_pages  
    c02cfe00 d real\_mode\_gdt\_entries  
    c02d028c D kstack\_depth\_to\_print  
    c03218c0 B irq\_err\_count  
    c03218c4 b root\_irq\_dir  
  
    其中每项有三部分组成：  
    第三部分是项目的名称  
    第一部分是项目在内核中的偏移量；  
    第二部分是项目的类型  各种类型的特征代码的意义  
        A：表示全局函数或变量  
            比如其中的agp\_frontend\_cleanup在原码中的定义为：  
            extern void agp\_frontend\_cleanup(void);  
        B：表示 静态/全局/原子 变量|数组  
            比如上面的irq\_err\_count在原码中的定义分别为：  
            extern atomic\_t irq\_err\_count;            //include\asm-i386\Hw\_irq.h  
            atomic\_t irq\_err\_count;                //arch\i386\kernel\Irq.c  
            static volatile unsigned long irq\_err\_count;    //arch\i386\kernel\Irq.c  
        b：表示静态结构体  
            比如上面例子中的root\_irq\_dir，在原码中的定义是：  
            static struct proc\_dir\_entry \* root\_irq\_dir;    //arch\i386\kernel\Irq.c  
        D：表示已经定义过初值的变量  
            比如上面例子中的kstack\_depth\_to\_print，在原码中的定义是：  
            int kstack\_depth\_to\_print = 24;            //arch\i386\kernel、Traps.c  
        d：表示已经定义了初值的变量/数组  
        R：表示定义的常量[全局]数组  
            比如上面例子中的ide\_pio\_timings，在原码中的定义是：