Linux系统下的开发基础

2013-04-18 00:17 by zmkeil, 382 阅读, 0 评论, 收藏, 编辑

针对linux的开发无非两种：用户级别、内核级别。用户级别的开发也就是应用程序的开发，各种各样的应用程序数不胜数，开发方法也多种多样，这里当然不会介绍应用程序的开发方法，而只是说明其底层基础，也就是应用程序是如何在操作系统上运行的，操作系统做了哪些我们看不见的工作，了解这些工作有时对开发应用程序会有帮助，尤其是最求性能时，更要结合OS底层的实现。

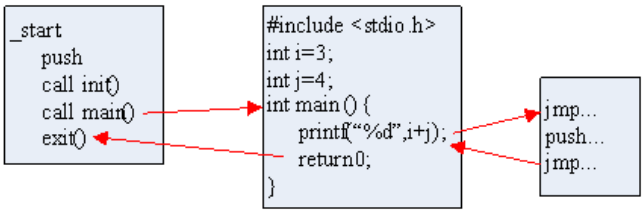
内核级别的开发相比较难些，一方面，linux社区的那些内核开发者们正在不断地更新完善内核，这是直接针对内核架构的开发；另一方面，linux内核的设计采用模块的机制，且模块可以自由加载、卸载，即可以为内核编写多种模块，并嵌入到内核中，最通常的就是设备驱动模块。这里仅简单介绍模块机制的实现原理。

1.应用程序运行

用高级语言（如最常见的c语言）开发应用程序，函数库是一个至关重要的内容，没有库，什么都自己裸写，将会很麻烦，甚至有些库（如某些CRT库）是程序编译链接时所必须的，没有库，根本就不能生成可执行文件。C语言有一个标准的库libc，里面包含了c中最常用的一些函数，如printf等。Linux下使用的是GNU开发的glibc库，和libc差不多，一些运行时函数可能不一样。

举一个最简单的例子，就这么一个简单的程序，编译链接好后也是很大的，其中至少用到两个库函数。一个是_start函数，它是CRT里的，它首先初始化一些全局变量，如i、j；然后调用主程序main函数；调用完后，它还要执行exit清理进程。Gcc编译链接器会默认吧这个库函数静态链接在可执行文件中的，并将它作为程序入口，即sys_execve()系统调用返回的地址。

另一个是printf函数，它是glibc中的，但它却是动态链接的，即在可执行文件中，并没有printf函数的实体，而只是一个jmp。



1.1动态链接库特性的实现

如果程序中全是静态链接好的库函数，则execve函数和之前将的linux0.11版本的execve函数没有太多区别，即打造好一个全新的进程空间，然后根据缺页中断实行按需加载。而现在有了动态链接库，execve函数需执行一些额外功能，操作系统也需提供一些新的服务。

先看动态链接库有哪些新特性。其一，它的实体不在可执行文件中，可执行文件中只保存它的符号名，并有一个jmp表来对应所有这些库函数调用；其二，和程序其它部分一样，它也是被按需加载的；其三，在一个运行的linux

About

昵称：zmkeil
园龄：3年3个月
粉丝：37
关注：0
+加关注

最新评论

Re:Luci实现框架
您好，想请教一个问题，我想将Luci的admin-full下面的syslog显示功能移植到admin-mini，请问怎么实现？ -- zyzferrari

日历

随笔分类

推荐排行榜

阅读排行榜

随笔档案

操作系统中，一个库函数只存在一个实体，内核维护着一个已加载的库函数表link_map，并为所有进程共享。

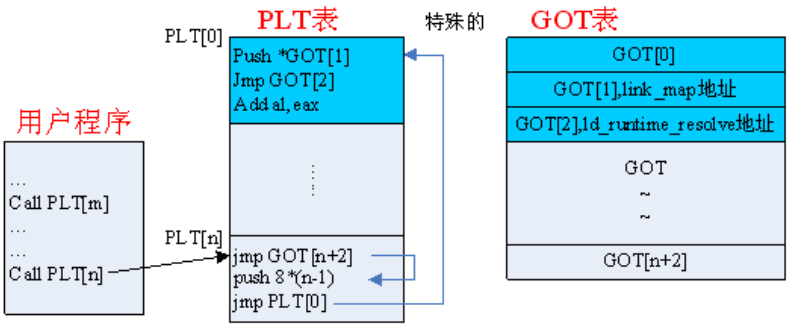
可见动态链接库方法，既节省磁盘空间，因为每个可执行文件可以很小；又节省内存空间，因为所有进程共享一个库函数实体。那么怎么实现上述的动态链接库功能呢？

首先看execve()函数，相比于linux0.11版的execve已经复杂多了，其具体的运行步骤如下：

1. sys_execve()执行一些检查，然后调用do_execve()；
2. do_exevce()读可执行头，比对魔数，判别是什么类型的文件；
3. 调用search_binary_handle()收索合适的处理函数，如elf格式的，就调用load_elf_binary()，执行如下：
    1. 检查elf格式；
    2. 寻找.interp段，设置动态连接器ld_so的路径；（动态链接器是一段内核程序，它可以根据动态库函数的符号名，将其实体入口插入到任意进程的调用处）
    3. 根据elf的programme_header，将文件加载到内存中；（这里只是加载一部分吗？难道按需加载策略不用了）
    4. 初始化进程环境。。。（同以前的execve的工作）
    5. 将sys_execve的返回地址改为elf文件的入口地址，即程序的_start处。
4. 之后load_elf_binary()返回，do_execve()返回，sys_execve()返回，到用户态执行elf入口

其中与动态链接库相关的最重要的一步是设置动态连接器的路径，个人理解就是把动态连接器的地址放在该进程的某个特殊位置，用户程序在执行过程中，若要调用某个动态链接库程序，就会执行到这个连接器程序，把库函数实体的地址映射过来。

目前linux下可执行文件，通过PLT、GOT表的方式来实现这点，具体如下图：



其中PLT表中存放的是一组代码，供用户程序调用动态库函数时调用，其中PLT[0]比较特殊。GOT表中存放的是一组地址，其中前三项比较特殊，GOT[1]存放的是本进程link_map的地址，由它结合库函数号就可以确定该库函数的符号名，然后ld程序就可以去内核维护的link_map中去找该库函数的实体了。GOT[2]存放的是连接器的地址，它初始是为0，sys_execve加载时，根据文件的类型，将它设置为相应连接器程序的入口。GOT[n+2]初始时存放的是PLT[n]段中第二条指令push的地址。具体的执行流程如下：

1. 用户程序中调用库函数call PLT[n]，首先jmp GOT[n+2]中存放的地址处，

3. Linux下的虚拟Bridge实现(3963)
4. OpenWRT平台搭建及简单应用(3162)
5. Linux下VLAN功能的实现(1967)

前面所述，它指向push 8*(n-1)，然后执行PLT[0]；

2. PLT[0]继续把link_map的指针地址push，作为参数调用GOT[2]，即ld_程序；
3. Ld_程序根据传入的两个参数，去内核的link_map中找相应的库函数实体，并将其入口写到GOT[n+2]中，然后重新跳转到PLT[n]处（省略若干细节）；
4. 再次执行jmp GOT[n+2]时就会跳入相应的库函数实体中，并且以后再调用这个库函数时，就可以直接进入了，因为GOT[n+2]写好后就不会变了，变的只是每次库函数的调用参数不同。

1.2一个编写调试的例子

如下图所示，是一个很简单的例程。现在要把addvec.c和multvec.c编译成一个动态链接库，具体用法如下：

```
/*addvec.c*/
void addvec() {...;...}

/*multvec.c*/
void multvec() {...;}
```

```
/*vector.h*/
void addvec();
void multvec();
```

```
/*main.c*/
#include "vector.h"
int main() {
    addvec();
    multvec();
}
```

制作动态库：gcc –o libvector.so –shared –fPIC addvec.c multvec.c，生成共享动态库文件libvector.so；

编译应用程序：gcc –c main.c，生成目标文件main.o；

链接应用程序：gcc –o main main.o –L. –lvector，生成可执行文件main。其中-L.告诉编译器在当前目录下找动态库，-lvector表示所要的库文件（-l后面的符号，在前面加上lib，后面加上.so即为库文件的名字，这是gcc默认的）。

运行程序：上面只是完成了编译链接的工作，程序执行时，内核必须能够找到库函数的所在（这些在编译链接好的可执行文件中是找不到的），因此必须先把该库文件所在目录放入内核默认搜索库文件的范围中去。可以用临时的export LD_LIBRARY_PATH=/home/...dir/，当然也可以把该目录写入/etc/ld.so.conf文件中去，重启即可作为永久的库文件搜索目录。

Linux下查看调试可执行文件的一个工具时objdump命令。简单介绍其基本用法：

查看文件section节：objdump –h main，可以看到可执行文件中的各个段，如got.plt，plt，text，interp等；

查看代码段：objdump –j text main，可以看到其中的库调用，如_start，init，printf等；

Gdb调试时，可根据前面查出的GOT表的位置，x/8x addr查看GOT的值y，并info symbol y就可以查看映射的库函数符号名了。

1.3小结

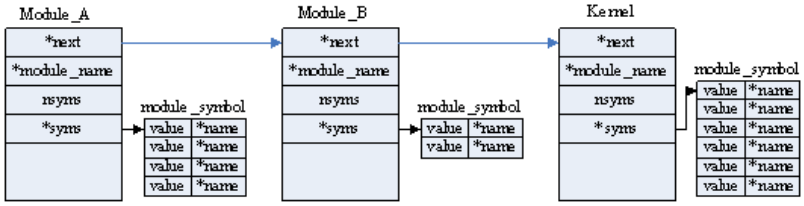
上面介绍了应用程序从编译到运行到动态加载的基本过程，当然其中很多细节都没细看，如ld_装载器如何工作等，那些都是内核来实现的，对于应用程

序而言不必要细究。却是内核为我们搭好了一个非常优秀的运行环境，对应用程序而言，只要关注与程序的逻辑实现就可以了，不过上述的一些基本流程还是很有必要的，起码知道动态库的优点缺点等。

2.模块机制

Linux内核采用模块的机制，即整个内核有多个模块组成，其中Kernel就是最原始，最重要的那个模块，而其它模块则可根据需要动态加载或卸载。它们共同组成一个内核整体，为用户提供服务。

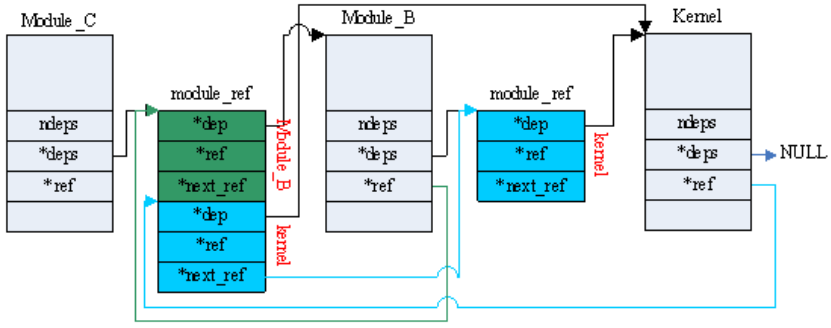
它们都是运行在内核空间内，之间的函数是可以相互调用的，那么新加载进来的模块怎么找到它所调用的其它模块的函数呢？原来每个模块都会把那些愿意被其它模块调用的函数及地址列成一张表，如下图所示：



首先看struct_module结构，其中module_name表示其名称，next构成一个列表，由一个全局的module_list变量即可索引所有模块，其中kernel在链表尾。各个模块导出自己的函数，比如Kernel模块就会导出printk()、copy_to_user()等函数，共其它模块使用。我们编写的模块也可以导出一些函数，不过如果相对独立则每必要那么做。

注意，module刚被加载时，它应该有一张需要引用的函数表，内核根据这个表去查所有模块的syms表，来找对应项，对对新模块做地址重定位。重定位好后引用函数表就可以扔了，但导出表要一直留着。

有上述可知，每个模块可能会依赖于其它模块，同时又被很多模块引用，怎么建立这种dep/ref表呢？如下图所示：



如上图所示，1）依赖关系很简单，struct_module结构中*deps指向该模块的依赖表module_ref，该依赖表中有很多项，每一项代表一个它依赖的模块，并有其中的*dep指向。2）依赖关系复杂一点，由struct_module结构中的ref指向引用它的最后一个模块的module_ref结构中代表它的那项。然后又该项中的next_ref指向下一个引用它的模块。如上图所示，蓝色线就代表了kernel模块的引用关系，墨绿色线就代表了module_B的引用关系。3）最后module_ref中每一项的ref指向自己（好像又不是，这项还没清楚其用途）。

之所以这样设计，因为依赖dep关系在模块加载完成后就确定了，是静态的，可以设计成静态表的结构，即每个模块有自己固定的module_ref；而引用关系则不能确定，而是可能随着新增模块而不断增加，因此设计成链表结

构，可以自由添加。

2.1加载卸载机制

模块的加载有两种方式，一种是手动通过insmod加载，也可以通过守护进程在系统启动时自动加载。最终的实现大同小异，下面主要看一下insmod()的实现。

1. 系统调用query_module()来遍历所有模块（利用module_list）中的*symbol表中的所有符号，与本模块中的引用外部符号表对比
    1. 用找到的符号的地址来对本模块进行重定位；
    2. 记录下依赖了哪些模块，并填写它的module_ref表中的dep项；
    3. 填写自身的module_symbol表；

    （注意此时该module还在用户空间，相当于一个文件，只是对其做了一些修改）
2. 系统调用create_module()，分析足够的内核空间，并初始化struct_module()；
3. 系统调用init_module()，将module复制到该内核空间来，当然地址信息都是计算好的；然后更新全局的依赖/引用关系；调用模块的初始化函数init_module()（如果有的话）。

    这样载入后，该module就和kernel同等了。

卸载很简单，rmmod module即可，内核会先检查它的struct_module中的ref指针，应该为空，表示没有其他模块引用它了，否则会出错；然后调用模块的退出程序exit_module()（如果有的话）；然后修改全局的dep/ref关系；在module_list链表中删除该项；最后释放内存。

2.2模块编写基础

编写一个最简单的模块，什么功能都不用实现，仅有一个init函数和exit函数，其中用到printk()是内核函数，表明我们的模块确实是运行在内核的，由此我们就可以说我们是linux内核开发者啦！哈哈。

```
/*hello.c*/
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
Static int __init hello_init() {
    printk("hello!\n");
    return0;
}
Static void __exit hello_exit() {
    printk("bye!\n");
}
Module_init(hello_init);
Module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("zmkeil")
```

```
/*Makefile*/
obj-m := hello.o
hello -objs := a.o,b.o

all:
(tab) make -C /lib/modules/$(shell uname -r) /build
M=$(shell pwd) modules

clean
(tab) make -C ..... clean
```

模块文件编写很简单，首先要包含3个最基本的头文件；最简单的模块只要包含两个基本函数init、exit即可，并由宏module_init()、module_exit()来标示；函数前的__init、__exit宏标示表示这个函数在一次使用完后，内核会将其释放，节省内存，是可选项；最后还有可选项来说明协议、作者等。

Makefile也很简单，obj-m表示编译成模块，-objs表示由哪些目标文件构成，make命令首先要找到内核源码中顶层的makefile文件，以便利用内核函数，并在当前目录下操作。

Make好后，该目录下有好几个文件hello.o，hello.ko，hello.mod.o，然后执行insmod hello.ko即可，要卸载时执行rmmod hello。

2.3设备驱动

设备驱动是最典型的模块，它要和设备联系在一起，相比于前面那个简单模块，它需要要增加一些内容。我们知道在linux系统中，每个设备被当成一个文件，一般在/dev/*目录下，其魔数magic会标示它是一个设备文件；读到这个设备文件时，内核实际上会去它所维护的设备表cdev[]/bdev[]中去寻找；最后才映射到相应的驱动程序来执行功能。

在/dev/目录下一sudo执行mknod命令创建设备文件，c表示是字符设备，MAJ 和 MINO分别表示主设备号和子设备号，主设备号会和cdev[]联系在一起，并映射到相应的设备驱动中去；那么设备驱动模块也必须与cdev[]联系在一起，所以设备驱动的init函数通常要调用register_chrdev_region(dev_t,10,proc_name)来注册设备。MINO号与cdev[]无关，只是传递给设备启动程序，来执行不同的功能。

设备驱动一般还会实现一些基本的功能函数，如read，write，ioctrl等，并由file_operations结构封装好。

设备驱动程序的init函数首先申请一项cdev[]，然后与两个东西绑定：一是前面那个file_operations结构，一是dev文件。

```
Static int zm_read(...){ ... }
Static int zm_write(...){ ... }
Static const struct file_operations char_ops={ .read=zm_read,
                                               write=zm_write,...};
Module_init() {
    dev_t dev=MKDEV(MAJOR,MINOR);
    register_chrdev_region(dev,10,proc_name);
或者用
    alloc_chrdev_region(*dev,…...); 动态申请设备号

    chardev=cdev_alloc(); 申请一个cdev[]项
    cdev_init(chardev,&char_ops); 与操作函数绑定在一起
    cdev_add(chardev,dev,1); 与设备号（即设备文件）绑定
}
```

2.4小结

模块机制是linux内核更加灵活，也会广大爱好者提供良好的内核开发的环境，特别是其中设备驱动开发，充满乐趣，以后还会专门学习。

3.总结

这些内容其实是和OS学习一起完成的，纸质笔记上最后的落款是2012.6.13—23:11，到现在都快半年了吧，呵呵！

Linux开发吧，应用程序，还有内核程序，暴风雨来得更猛烈些吧！

赵莽

2012.11.14

zmkeil
关注 - 0
粉丝 - 37
+加关注

0 0

« 上一篇：STRAIGHT关键技术研究
» 下一篇：现代计算机体系架构带来的挑战

分类：操作系统

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。