

內核調試一般比應用層程序調試困難很多，主要因為內核是系統開機後執行的第一個程序，這使得用於一般應用程序的調試方法無法使用在內核上。自由軟件社群開發了一些特殊方法解決這問題。這些方法中最方便的方法莫過於KGDB 的使用。KGDB 是個一特殊的內核輔助工具，除了在內核代碼中加入了一些調試代碼外也提供一個gdbstub 用於和遠程gdb 調試程序聯機用。以前，這樣一個使用遠程gdb 調試內核的開發需要在一般linux內核上打KGDB 補丁(patches)同時編譯時使用特殊編譯設置來完成。可喜的是，至linux-2.6.xx（xx多少記不清了）後的版本內核已經正式將kgdb 加入為主流核心發布的一部份。換句話說，內核開發者幾乎不需花任何額外的功夫就可使用kgdb。此外，kgdb 成為主流內核發行一部份也代表他的穩定性及實用性受到社群的肯定。

由於kgdb的方便易用，大大提高了linux平台下驅動開發者的效率。本文以及後面一系列文章從最基本的開始詳細描述瞭如何搭建一個linux驅動調試環境。如何加載模塊開始調試內核模塊，如何調試模塊的初始化函數。

kvm系統的前端是qemu-kvm，工作在用戶空間，給用戶提供一套方便的kvm虛擬化工具集合。下面來介紹一下qemu-kvm-0.11.0的編譯過程。

1、下載

[plain] view plain copy print ? 

1. `wget`
`http://sourceforge.net/projects/kvm/files/qemu-kvm/0.11.0/qemu-kvm-0.11.0.tar.gz/download`
- 2.
3. `mv download qemu-kvm-0.11.0.tar.gz`
- 4.
5. `tar -xzf qemu-kvm-0.11.0.tar.gz`

2、安裝其他庫

[plain] view plain copy print ?  

1. `sudo apt-get install libpci-dev`
- 2.
3. `sudo apt-get install libsd11.2-dev`

3、編譯安裝qemu

`cd qemu-kvm-0.11.0`

`./configure`

`make`

`sudo make install`

4、將qemu 添加到環境變量

進入目錄 `/usr/local/bin`

為qemu-system-x86_64 創建符號鏈接：

`$ sudo ln -s qemu-system-x86_64 qem`

打開/etc/profile 文件在末尾添加

`PATH="$PATH:/usr/local/bin"`

`export PATH`

使環境變量生效，在終端輸入：

`source /etc/profile`

另外打開一個終端輸入qemu 可見可以正常啟動。

但是發現這時提示找不到kvm。

Ubuntu10.10 系統有自帶的kvm 內核模塊。

打開文件/etc/modules

在其中加入想要加載的模塊名

kvm

kvm-amd

這樣在再次重啟的時候會自動加載模塊 **kvm** **kvm-amd**

注意不要加擴展名.ko

linux內核調試環境搭建-2 用busybox搭建

下載linux內核：

```
$cd ~/work/
```

```
$wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.35.9.tar.bz2
```

解壓

```
$tar -jxvf linux-2.6.35.9.tar.bz2
```

拷貝現有系統的內核配置選項

```
$ cp /boot/config-2.6.35-22-generic .config
```

確保如下選項被選中：

在這裡建議關閉一個選項：DEBUG_RODATA CONFIG_DEBUG_RODATA = n 該選項是將內核的一些內存區域空間設置為只讀，這樣可能導致kgdb 的設置軟斷點功能失效。所以推薦將該選項關閉。

Location:

-> Kernel hacking ->Write protect kernel read-only data structures

另外還得關閉掉一個選項：CONFIG_DEBUG_SET_MODULE_RONX

location：

Kernel hacking --->

Set loadable kernel module data as NX and text as RO

關於這個選項的意義不多說，還是看註釋吧：

[plain] view plain copy print ?  

1. This option helps catch unintended modifications to loadable
2. kernel module's text and read-only data. It also prevents execution
3. of module data. Such protection may interfere with run-time code
4. patching and dynamic kernel tracing - and they might also protect
5. against certain classes of kernel exploits.
6. If in doubt, say "N".

那些搞linux內核rootkit的則要注意這個選項了！！。不知能不能在開啟這個選項的時候讓其失效？

上面兩個選項都要去掉，有一個選擇上那麼在調試內核模塊的時候都會導致下斷點錯誤：

[plain] view plain copy print ?  

1. Cannot insert breakpoint 4.
2. Error accessing memory address 0xd0ce8000: 未知的錯誤4294967295

KGDB_SERIAL_CONSOLE 打開該選項: 使用串口進行通信

位置：> Kernel hacking

-> KGDB: kernel debugger

-> KGDB: use kgdb over the serial console

KGDB_LOW_LEVEL_TRAP 使能該選項可以kgdb 不依賴notifier_call_chain() 機制來

獲取斷點異常, 這樣就可以對notifier_call_chain() 機制實現相關的函數進行單步調試。

依賴: KGDB [=y] && (X86 [=y] || MIPS [=MIPS])

位置: -> Kernel hacking

-> KGDB: kernel debugger (KGDB [=y])

->KGDB: Allow debugging with traps in notifiers

DEBUG_INFO 該選項可以使得編譯的內核包含一些調試信息，使得調試更容易。

位置：

-> Kernel hacking

->compile the kernel with debuginfo

FRAME_POINTER 使能該選項將使得內核使用幀指針寄存器來維護堆棧，從而就可以正確地執行堆棧回溯，即函數調用棧信息。

位置:

-> Kernel hacking

->Compile the kernel with frame pointers

MAGIC_SYSRQ (如果你選擇了KGDB_SERIAL_CONSOLE, 這個選項將自動被選上) 激活" 魔術 SysRq" 鍵. 該選項對kgdboc 調試非常有用，kgdb 向其註冊了'g' 魔術鍵來激活kgdb。

位置:

-> Kernel hacking

->magic SysRq key

當你想手動激活kgdb 時，你可以觸發SysRq 的g 鍵, 如:

```
$ echo "g" > /proc/sysrq-trigger
```

CONFIG_8139CP 選擇這個選項來驅動qemu-kvm的網卡設備，以備以後使用。

位置：

Device Drivers

Network device support

Ethernet (10 or 100Mbit)

RealTek RTL-8139 C+ PCI Fast Ethernet Adapter support (EXPERIMENTAL)

修改編譯優化等級:

打開根目錄下Makefile 文件修改內核Makefile 優化選項將KBUILD_CFLAGS +=
-O2, 修改 為：KBUILD_CFLAGS += -O 。

編譯內核:

make && make modules

編譯完成後，複製bzImage 和vmlinix 到工作目錄下備用

```
cp arch/x86/boot/bzImage .
```

```
cp vmlinix .
```

編譯busybox：

```
cd ~/work/busy/
```

下載busybox-1.20.1.tar.bz2:

wget http://www.busybox.net/downloads/busybox-1.20.1.tar.bz2

解壓縮：

tar -jxvf busybox-1.20.1.tar.bz2

進入busybox 目錄：

\$ cd busybox-1.20.1/

編譯busybox

make menuconfig

Busybox Settings --->

Build Options --->

☒ Build BusyBox as a static binary (no shared libs)

Installation Options --->

☒ Don't use /usr

Miscellaneous Utilities --->

☐ flashcp

☐ flash_lock

☐ flash_unlock

☐ flash_eraseall

注:[] 表示不選擇

保存配置文件後開始編譯和安裝

make

```
make install
```

此時在當前目錄下生成了一個_install 目錄，裡面就是busybox 的執行文件

製作文件系統

使用如下命令來創建一個虛擬文件系統磁盤文件，

在當前目錄下創建一個名為busybox.img, 大小為300M 的文件，並將其格式化為ext3 的文件系統

```
dd if=/dev/zero of=./busybox.img bs=1M count=300
```

```
mkfs.ext3 busybox.img
```

將這個虛擬磁盤文件到本地系統中，這樣我們可以像訪問本地文件一樣訪問它，並將生成好的busybox 的文件拷貝到這個文件裡。

```
sudo mkdir /mnt/disk
```

```
sudo mount -o loop busybox.img /mnt/disk
```

```
sudo cp -rf /dir/to/busybox-1.17.0/_install/* /mnt/disk
```

創建必須的文件系統目錄

```
cd /mnt/disk/
```

```
sudo mkdir dev sys proc etc lib mnt
```

創建設備節點

```
$ cd dev/
```

```
$sudo mknod console c 5 1
```

```
$sudo mknod null c 1 3
```

```
$sudo mknod tty2 c 4 2
```


使用busybox 默認的設置文件

```
sudo cp -a /dir/to/busybox-1.17.0/examples/bootfloppy/etc/* /mnt/disk/etc
```

```
sudo vi /mnt/disk/etc/init.d/rcS
```

將下面內容拷貝到rcS 裡:

[plain] view plain copy print ?  

```
1.  #! /bin/sh
2.
3.  MAC=08:90:90:59:62:21
4.
5.  IP=192.168.100.2
6.
7.  Mask=255.255.255.0
8.
9.  Gateway=192.168.100.1
10.
11.
12.
13. /sbin/ifconfig lo 127.0.0.1
14.
15. ifconfig eth0 down
16.
17. ifconfig eth0 hw ether $MAC
18.
19. ifconfig eth0 $IP netmask $Mask up
20.
21. route add default gw $Gateway
22.
23.
24.
25. /bin/mount -a
26.
27. /bin/mount -t sysfs sysfs /sys
28.
29. /bin/mount -t tmpfs tmpfs /dev
```

```
30.  
31. /sbin/mdev -s  
32.  
33. mount -o remount,rw,noatime -n /dev/root /
```

修改fstab 文件：

[plain] view plain copy print ?  

```
1. $ vim etc/fstab  
2. proc /proc proc defaults 0 0  
3. tmpfs /tmp tmpfs defaults 0 0  
4. sysfs /sys sysfs defaults 0 0  
5. tmpfs /dev tmpfs defaults 0 0  
6. var /dev tmpfs defaults 0 0
```

做完上面對工作後，我們就可以卸載虛擬磁盤文件了。

```
sudo umount /mnt/disk
```

使用qemu 運行自己編譯的內核

```
qemu -m 512 -kernel bzImage -append "root=/dev/sda" -boot c -hda busybox.img  
-k en-us
```

運行如下圖所示：

```
QEMU
[ 0.921006] md: Waiting for all devices to be available before autodetect
[ 0.921519] md: If you don't use raid, use raid=noautodetect
[ 0.922117] md: Autodetecting RAID arrays.
[ 0.922551] md: Scanned 0 and added 0 devices.
[ 0.922995] md: autorun ...
[ 0.923387] md: ... autorun DONE.
[ 0.927606] EXT3-fs: barriers not enabled
[ 0.928412] kjournald starting. Commit interval 5 seconds
[ 0.928948] EXT3-fs (sda): mounted filesystem with ordered data mode
[ 0.929649] UFS: Mounted root (ext3 filesystem) readonly on device 8:0.
[ 0.930675] devtmpfs: mounted
[ 0.931098] Freeing unused kernel memory: 604k freed
ifconfig: SIOCGIFFLAGS: No such device
ifconfig: SIOCSIFHWADDR: No such device
ifconfig: SIOCSIFADDR: No such device
route: SIOCADDRT: No such process
Processing /etc/profile... Done
/ # [ 1.020079] Clocksource tsc unstable (delta = -137430807 ns)
/ # ls
bin      etc      linuxrc  mnt      rw.sh    sys
dev      lib      lost+found  proc    /sbin    usr
/ #
```

讓qemu與主機之間可以ping通：

在主機放入一腳本nettap.sh：

\$cd ~

\$touch nettap.sh

\$gedit nettap.sh

腳本中放入如下內容：

[plain] view plain copy print ?

1. tuncctl -u gdujian -t tap0
2. ifconfig tap0 192.168.100.1 up
3. echo 1 > /proc/sys/net/ipv4/ip_forward
4. iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
5. iptables -I FORWARD 1 -i tap0 -j ACCEPT
6. iptables -I FORWARD 1 -o tap0 -m state --state RELATED,ESTABLISHED -j ACCEPT

開機後用root權限執行這個腳本。

安裝tuncctl 工具

[plain] view plain copy print ?

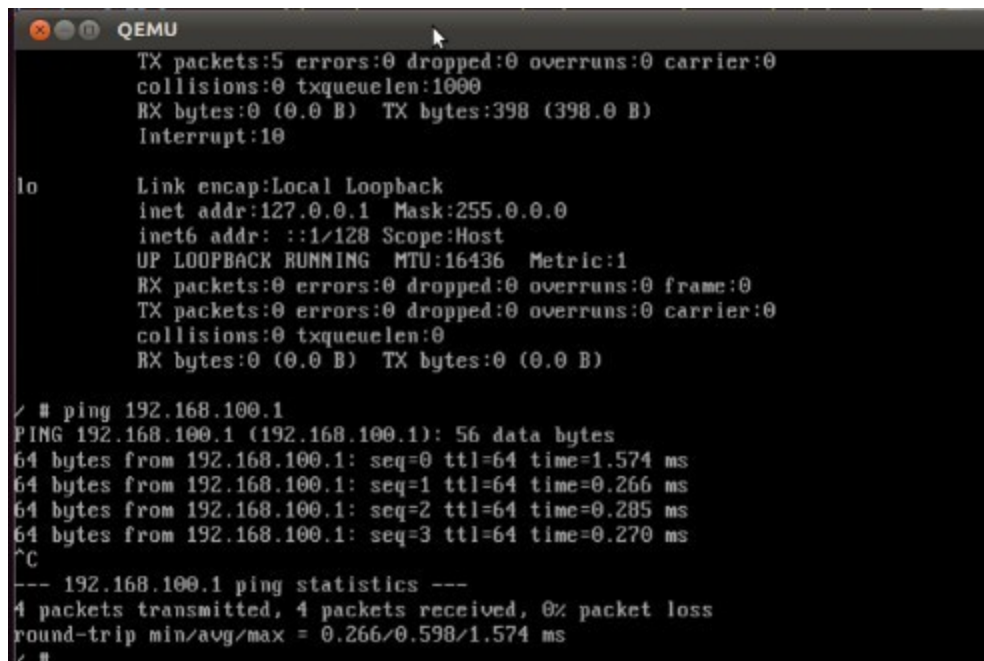
```
1. sudo apt-get install uml-utilities
```

用如下腳本啟動虛擬機：

```
qemu -m 512 -kernel bzImage -append "root=/dev/sda" -boot c -hda busybox.img -k en-us
```

```
-net nic -net tap,ifname=tap0,script=no
```

在虛擬機中ping主機可見可以ping的通：



```
QEMU
TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:398 (398.0 B)
Interrupt:10

lo    Link encap:Local Loopback
      inet addr:127.0.0.1 Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
      UP LOOPBACK RUNNING MTU:16436 Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

/ # ping 192.168.100.1
PING 192.168.100.1 (192.168.100.1): 56 data bytes
64 bytes from 192.168.100.1: seq=0 ttl=64 time=1.574 ms
64 bytes from 192.168.100.1: seq=1 ttl=64 time=0.266 ms
64 bytes from 192.168.100.1: seq=2 ttl=64 time=0.285 ms
64 bytes from 192.168.100.1: seq=3 ttl=64 time=0.270 ms
^C
--- 192.168.100.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.266/0.598/1.574 ms
/ #
```

讓qemu與主機共享文件（通過tftp方式）：

首先在ubuntu10.10中搭建tftp服務：

<http://blog.csdn.net/xsckernel/article/details/8521873>

只是在busybox中使用tftp客戶端命令有些不一樣。見後面介紹。

使用gdb + kgdb調試內核

使能kgdb可以在內核啟動時增加使能參數，這裡我們採取在內核啟動時增加啟動參數

(kgdboc=ttyS0,115200 kgdbwait)的方式：

```
qemu -m 512 -kernel bzImage -append "root=/dev/sda kgdboc=ttyS0,115200
kgdbwait" -boot c -hda busybox.img -k en-us -serial tcp::4321,server
```

這時，運行qemu的終端將提示等待遠程連接到本地端口4321：

```
QEMU waiting for connection on: tcp:0.0.0.0:4321,server
```

這時使用另外一個控制台執行：

```
gdb vmlinux
(gdb) target remote localhost:4321
```

然後qemu就可以繼續正常運行下去，最後停止內核，並顯示如下信息：

```
kgdb: Waiting for connection from remote gdb...
```

這時gdb這邊就可以看到如下的提示：

```
(gdb) target remote localhost:4321
Remote debugging using localhost:4321
kgdb_breakpoint () at kernel/debug/debug_core.c:983
983 wmb(); /* Sync point after breakpoint */
(gdb)
```

開始你的內核之旅吧～～～

如果gdb提示如下信息：

```
warning: Invalid remote reply:
```

可以使用Ctrl+C來終止當前gdb的操作，再次使用下面命令重新連接一次kgdb即可：

(gdb) target remote localhost:4321

linux內核調試環境搭建-3 調試內核模塊

在虛擬機中 創建兩個腳本get.sh:

[plain] view plain copy print ?

```
1. tftp 192.168.100.1 -g -r $1
```

put.sh:

[plain] view plain copy print ?

```
1. tftp 192.168.100.1 -p -l $1
```

在linux設備驅動開發詳解中源碼：

[cpp] view plain copy print ?

```
1.  /*=====
2.     A globalmem driver as an example of char device drivers
3.
4.     The initial developer of the original code is Baohua Song
5.     <author@linuxdriver.cn>. All Rights Reserved.
6.  ===== */
7. #include <linux/module.h>
8. #include <linux/types.h>
9. #include <linux/fs.h>
10. #include <linux/errno.h>
11. #include <linux/mm.h>
12. #include <linux/sched.h>
13. #include <linux/init.h>
14. #include <linux/cdev.h>
15. #include <asm/io.h>
16. #include <asm/system.h>
```

```

17. #include <asm/uaccess.h>
18. #include <linux/slab.h>
19.
20. #define GLOBALMEM_SIZE 0x1000 /*全局內存最大4K字節*/
21. #define MEM_CLEAR 0x1 /*清除全局內存*/
22. #define GLOBALMEM_MAJOR 245 /*預設的globalmem的主設備號*/
23.
24. static globalmem_major = GLOBALMEM_MAJOR;
25. /*globalmem設備結構體*/
26. struct globalmem_dev
27. {
28.     struct cdev cdev; /*cdev結構體*/
29.     unsigned char mem[GLOBALMEM_SIZE]; /*全局內存*/
30. };
31.
32. struct globalmem_dev *globalmem_devp; /*設備結構體指針*/
33. /*文件打開函數*/
34. int globalmem_open( struct inode *inode, struct file *filp)
35. {
36.     /*將設備結構體指針賦值給文件私有數據指針*/
37.     filp->private_data = globalmem_devp;
38.     return 0;
39. }
40. /*文件釋放函數*/
41. int globalmem_release( struct inode *inode, struct file *filp)
42. {
43.     return 0;
44. }
45.
46. /* ioctl設備控制函數 */
47. static int globalmem_ioctl( struct inode *inodep, struct file *filp, unsigned
48. int cmd, unsigned long arg)
49. {
50.     struct globalmem_dev *dev = filp->private_data; /*獲得設備結構體指針*/
51.
52.     switch (cmd)
53.     {
54.         case MEM_CLEAR:
55.             memset(dev->mem, 0, GLOBALMEM_SIZE);

```

```

56.     printk(KERN_INFO "globalmem is set to zero\n" );
57.     break ;
58.
59.     default :
60.         return  - EINVAL;
61. }
62. return  0;
63. }
64.
65. /*讀函數*/
66. static ssize_t globalmem_read( struct file *filp, char __user *buf, size_t size,
67.     loff_t *ppos)
68. {
69.     unsigned long p = *ppos;
70.     unsigned int count = size;
71.     int ret = 0;
72.     struct globalmem_dev *dev = filp->private_data; /*獲得設備結構體指針*/
73.
74.     /*分析和獲取有效的寫長度*/
75.     if (p >= GLOBALMEM_SIZE)
76.         return count ? - ENXIO: 0;
77.     if (count > GLOBALMEM_SIZE - p)
78.         count = GLOBALMEM_SIZE - p;
79.
80.     /*內核空間->用戶空間*/
81.     if (copy_to_user(buf, ( void *)(dev->mem + p), count))
82.     {
83.         ret = - EFAULT;
84.     }
85.     else
86.     {
87.         *ppos += count;
88.         ret = count;
89.
90.         printk(KERN_INFO "read %d bytes(s) from %d\n" , count, p);
91.     }
92.
93.     return  ret;
94. }

```



```

95.
96. /*寫函數*/
97. static ssize_t globalmem_write( struct file *filp, const char __user *buf,
98.     size_t size, loff_t *ppos)
99. {
100.     unsigned long p = *ppos;
101.     unsigned int count = size;
102.     int ret = 0;
103.     struct globalmem_dev *dev = filp->private_data; /*獲得設備結構體指針*/
104.
105.     /*分析和獲取有效的寫長度*/
106.     if (p >= GLOBALMEM_SIZE)
107.         return count ? - ENXIO: 0;
108.     if (count > GLOBALMEM_SIZE - p)
109.         count = GLOBALMEM_SIZE - p;
110.
111.     /*用戶空間->內核空間*/
112.     if (copy_from_user(dev->mem + p, buf, count))
113.         ret = - EFAULT;
114.     else
115.     {
116.         *ppos += count;
117.         ret = count;
118.
119.         printk(KERN_INFO "written %d bytes(s) from %d\n" , count, p);
120.     }
121.
122.     return ret;
123. }
124.
125. /* seek文件定位函數 */
126. static loff_t globalmem_llseek( struct file *filp, loff_t offset, int orig)
127. {
128.     loff_t ret = 0;
129.     switch (orig)
130.     {
131.         case 0: /*相對文件開始位置偏移*/
132.             if (offset < 0)
133.                 {

```

```

134.         ret = - EINVAL;
135.         break ;
136.     }
137.     if ((unsigned int )offset > GLOBALMEM_SIZE)
138.     {
139.         ret = - EINVAL;
140.         break ;
141.     }
142.     filp->f_pos = (unsigned int )offset;
143.     ret = filp->f_pos;
144.     break ;
145. case 1:     /*相對文件當前位置偏移*/
146.     if ((filp->f_pos + offset) > GLOBALMEM_SIZE)
147.     {
148.         ret = - EINVAL;
149.         break ;
150.     }
151.     if ((filp->f_pos + offset) < 0)
152.     {
153.         ret = - EINVAL;
154.         break ;
155.     }
156.     filp->f_pos += offset;
157.     ret = filp->f_pos;
158.     break ;
159. default :
160.     ret = - EINVAL;
161.     break ;
162. }
163. return ret;
164. }
165.
166. /*文件操作結構體*/
167. static const struct file_operations globalmem_fops =
168. {
169.     .owner = THIS_MODULE,
170.     .llseek = globalmem_llseek,
171.     .read = globalmem_read,
172.     .write = globalmem_write,

```

```

173.     .ioctl = globalmem_ioctl,
174.     .open = globalmem_open,
175.     .release = globalmem_release,
176. };
177.
178. /*初始化並註冊cdev*/
179. static void globalmem_setup_cdev( struct globalmem_dev *dev, int index)
180. {
181.     int err, devno = MKDEV(globalmem_major, index);
182.
183.     cdev_init(&dev->cdev, &globalmem_fops);
184.     dev->cdev.owner = THIS_MODULE;
185.     dev->cdev.ops = &globalmem_fops;
186.     err = cdev_add(&dev->cdev, devno, 1);
187.     if (err)
188.         printk(KERN_NOTICE "Error %d adding LED%d" , err, index);
189. }
190.
191. /*設備驅動模塊加載函數*/
192. int globalmem_init( void )
193. {
194.     int result;
195.     dev_t devno = MKDEV(globalmem_major, 0);
196.
197.     /* 申請設備號*/
198.     if (globalmem_major)
199.         result = register_chrdev_region(devno, 1, "globalmem" );
200.     else /*動態申請設備號*/
201.     {
202.         result = alloc_chrdev_region(&devno, 0, 1, "globalmem" );
203.         globalmem_major = MAJOR(devno);
204.     }
205.     if (result < 0)
206.         return result;
207.
208.     /* 動態申請設備結構體的內存*/
209.     globalmem_devp = kmalloc( sizeof ( struct globalmem_dev), GFP_KERNEL);
210.     if (!globalmem_devp) /*申請失敗*/
211.     {

```

```

212.         result = - ENOMEM;
213.         goto fail_malloc;
214.     }
215.     memset(globalmem_devp, 0, sizeof ( struct globalmem_dev));
216.
217.     globalmem_setup_cdev(globalmem_devp, 0);
218.     return 0;
219.
220. fail_malloc: unregister_chrdev_region(devno, 1);
221.     return result;
222. }
223.
224. /*模塊卸載函數*/
225. void globalmem_exit( void )
226. {
227.     cdev_del(&globalmem_devp->cdev);    /*註銷cdev*/
228.     kfree(globalmem_devp);    /*釋放設備結構體內存*/
229.     unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*釋放設備號*/
230. }
231.
232. MODULE_AUTHOR( "Song Baohua" );
233. MODULE_LICENSE( "Dual BSD/GPL" );
234.
235. module_param(globalmem_major, int , S_IRUGO);
236.
237. module_init(globalmem_init);
238. module_exit(globalmem_exit);

```

其makefile文件：

[plain] view plain copy print ?

```

1. obj-m += globalmem.o
2. KDIR = /home/gudujian/work/linux-2.6.35.9
3.
4. EXTRA_CFLAGS=-g -O0
5.
6. build:kernel_modules
7.
8. kernel_modules:

```

```

9.      make -C $(KDIR) M=$(CURDIR) modules
10.
11.
12. clean:
13.      make -C $(KDIR) M=$(CURDIR) clean

```

其中KDIR為編譯內核時使用的目錄。

腳本section.sh內容：

[plain] view plain copy print ?

```

1. #
2. # gdbline module image
3. #
4. # Outputs an add-symbol-file line suitable for pasting into gdb to examine
5. # a loaded module.
6. #
7. cd /sys/module/$1/sections
8. echo -n add-symbol-file `/bin/cat .text`
9.
10. for section in .[az]* *; do
11.     if [ $section != ".text" ]; then
12.         echo " \\"
13.         echo -n " -s" $section `/bin/cat $section`
14.     fi
15. done
16. echo

```

將得到的文件編譯結果，globalmem.ko；以及腳本section.sh 通過tftp方式拷貝到工作目錄：

#./get.sh globalmem.ko

#./get.sh section.sh

在主機的tftpboot目錄下創建一個文件gdb，權限777.

用如下腳本啟動虛擬機：

[plain] view plain copy print ?

```

1. qemu -m 512 -kernel bzImage -append "root=/dev/sda kgdboc=ttyS0,115200 kgdbwait" -boot

```

```
c -hda busybox.img -k en-us -net nic -net tap,ifname=tap0,script=no -serial  
tcp::4321,server
```

另開一個終端：

```
$cd /dir/to/linux-2.6.35.9
```

```
$gdb vmlinux
```

顯示如下：

```
Reading symbols from /home/gudujian/work/linuxker/linux-2.6.35.9/vmlinux...done.
```

```
(gdb)
```

gdb命令

```
(gdb) target remote localhost:4321
```

```
Remote debugging using localhost:4321
```

```
kgdb_breakpoint (new_dbg_io_ops=0xc07c27e0) at kernel/debug/debug_core.c:967
```

```
warning: Source file is more recent than executable.
```

```
967 wmb(); /* Sync point after breakpoint */
```

在主機終端按c讓qemu虛擬機啟動運行：

在qemu的虛擬機中加載模塊globalmem.ko

```
#insmod globalmem.ko
```

用section.sh腳本得到gdb符號文件：

```
#!/section.sh globalmem > gdb
```

將gdb符號文件拷貝到主機中：

```
#!/put.sh gdb
```

讓虛擬機進入調試模式：

```
#echo g >/proc/sysrq-trigger
```

/tftpboot/gdb 修改前後的內容分別是：

[plain] view plain copy print ?

```
1. add-symbol-file 0xe0a35000 \  
2.      -s .bss 0xe0a35834 \  
3.      -s .data 0xe0a356b8 \  
4.      -s .gnu.linkonce.this_module 0xe0a356c0 \  
5.      -s .note.gnu.build-id 0xe0a35540 \  
6.      -s .rodata 0xe0a35580 \  
7.      -s .strtab 0xe0a38430 \  
8.      -s .symtab 0xe0a38000 \  
9.      -s __mcount_loc 0xe0a35690 \  
10.     -s __param 0xe0a3567c
```

[plain] view plain copy print ?

```
1. add-symbol-file /dir/to/globalmem.ko 0xe0a35000 \  
2.      -s .bss 0xe0a35834 \  
3.      -s .data 0xe0a356b8 \  
4.      -s .gnu.linkonce.this_module 0xe0a356c0 \  
5.      -s .note.gnu.build-id 0xe0a35540 \  
6.      -s .rodata 0xe0a35580 \  
7.      -s .strtab 0xe0a38430 \  
8.      -s .symtab 0xe0a38000 \  
9.      -s __mcount_loc 0xe0a35690 \  
10.     -s __param 0xe0a3567c
```

此時在調試端輸入命令：

(gdb) source /tftpboot/gdb

下兩個斷點：

(gdb) b globalmem_write

Breakpoint 1 at 0xe0a351cf: file /dir/to/globalmem.c, line 100.

(gdb) b globalmem_read

Breakpoint 2 at 0xe0a350fc: file /dir/to/globalmem.c, line 100.

然後c讓qemu運行。

在qemu中創建一個設備節點globalmem：

```
#mknod /dev/globalmem c 245 0
```

（這裡的主設備號跟源代碼裡的相同）

在qemu中給節點/dev/globalmem輸入hello driver world:

```
#echo 「hello driver world」 > /dev/globalmem
```

此時主機中斷在globalmem_write

```
(gdb) c
```

Continuing.

Breakpoint 1, globalmem_write (filp=0xdfa96080,

buf=0x854c740 "hello driver world\n", size=19, ppos=0xdfbcbf98)

at /home/gudujian/06/globalmemDriver/globalmem.c:100

```
100 unsigned long p = *ppos;
```

此時查看變量：

```
(gdb) p buf
```

```
$3 = 0x854c740 "hello driver world\n"
```

```
(gdb) p /x size
```

```
$4 = 0x13 //字符串長度
```

```
(gdb) p *ppos
```

```
$5 = 0
```

如果有興趣可往下跟蹤，這裡略去，直接c了。

同理也可以用同樣的方式來調試內核模塊的其它函數。

linux內核調試環境搭建-4 調試模塊初始化函數

打開一終端執行：

[plain] view plain copy print ?

```
1. qemu -m 512 -kernel bzImage -append "root=/dev/sda kgdboc=ttyS0,115200 kgdbwait" -boot  
c -hda busybox.img -k en-us -net nic -net tap,ifname=tap0,script=no -serial  
tcp::4321,server
```

顯示等待調試端鏈接：

QEMU waiting for connection on: tcp:0.0.0.0:4321,server ◦

再打開一終端

\$cd linux-2.6.35.9

\$gdb vmlinux

調試客戶端：

(gdb) target remote localhost:4321

Remote debugging using localhost:4321

kgdb_breakpoint (new_dbg_io_ops=0xc07c27e0)

at kernel/debug/debug_core.c:967

warning: Source file is more recent than executable.

967 wmb(); /* Sync point after breakpoint */

開啟運行：

(gdb) c

Continuing.

一直到虛擬機啟動。

讓虛擬機進入調試模式：

```
#echo g > /proc/sysrq-trigger
```

在調試端下斷點：

```
(gdb)b sys_init_module
```

（這個函數名在源文件中找不到，是由宏定義SYSCALL_DEFINE3展開的）

讓虛擬機開啟運行：

```
(gdb) c
```

Continuing.

在虛擬機中加載模塊：

```
#insmod globalmem.ko
```

此時調試端中斷在函數sys_init_module中：

```
Breakpoint 1, sys_init_module (umod=0xb777c008, len=134933, uargs=0x81c4348 "")
```

```
    at kernel/module.c:2618
```

```
2618 if (!capable(CAP_SYS_MODULE) || modules_disabled)
```

下斷：

```
(gdb) b add_sect_attrs
```

```
Breakpoint 2 at 0xc017251d: file kernel/module.c, line 1160.
```

執行：

```
(gdb) c
```

Continuing.

```
Breakpoint 2, add_sect_attrs (mod=0xe0a356c0, nsect=34, secstrings=0xe0a2b621 "",
```

```
    sechdrs=0xe0a2b754) at kernel/module.c:1160
```

```
1160 for (i = 0; i < nsect; i++)
```

用where顯示調用棧：

(gdb) where

#0 add_sect_attrs (mod=0xe0a356c0, nsect=34, secstrings=0xe0a2b621 "",
sechdrs=0xe0a2b754)

at kernel/module.c:1160

#1 0xc0174029 in load_module (umod=<value optimized out>, len=<value optimized out>,
uargs=<value optimized out>) at kernel/module.c:2552

#2 0xc01742e2 in sys_init_module (umod=0xb78b6008, len=134933, uargs=0x81c4348 "")
at kernel/module.c:2622

下斷第1192行：

(gdb) b 1192

Breakpoint 4 at 0xc01725db: file kernel/module.c, line 1192.

按c開啟運行：

(gdb) c

Continuing.

Breakpoint 4, add_sect_attrs (mod=<value optimized out>, nsect=<value optimized out>,
secstrings=<value optimized out>, sechdrs=0xe0a2b754) at kernel/module.c:1192

1192 *(gattr++) = &(sattr++)->matr.attr;

此時打印 sattr ->name：

(gdb) p sattr ->name

\$4 = 0xdf412b60 ".note.gnu.build-id"

按若干次c之後：

```
(gdb) p sattr ->name
```

```
$2 = 0xdf8b25e8 ".text "
```

```
(gdb) p /x sattr ->address
```

```
$4 = 0xe0a35000
```

```
(gdb) p sattr ->name
```

```
$8 = 0xdf8b2600 ".data"
```

```
(gdb) p /x sattr ->address
```

```
$10 = 0xe0a356b8
```

在gdb中增加調試信息：

[plain] view plain copy print ?

```
1. add-symbol-file /home/gudujian/06/globalmemDriver/globalmem.ko 0xe0a35000 -s .data
    0xe0a356b8
```

此時就可以對globalmem.ko模塊中的符號，正常下斷點：

（這裡僅僅使用模塊的兩個節.data，.text如果再使用其它節那麼比較麻煩，如果有腳本可以做上面的事情就很好了）。

```
(gdb) b globalmem_init
```

```
Breakpoint 5 at 0xe0a35448: file /home/gudujian/06/globalmemDriver/globalmem.c, line
195.
```

按c讓程序開始運行：

(gdb) c

Continuing.

Breakpoint 5, globalmem_init () at /home/gudujian/06/globalmemDriver/globalmem.c:195

```
195 dev_t devno = MKDEV(globalmem_major, 0);
```

此時程序中斷在新加載模塊的module_init函數中：globalmem_init。

此時因為編譯這個內核模塊時帶有調試信息 EXTRA_CFLAGS=-g -O0

所以可以正常調試linux模塊。