目錄[-]

內核開發比用戶空間開發更難的一個因素就是內核調試艱難。內核錯誤往往會導致系統宕機，很難保留出錯時的現場。調試內核的關鍵在於你的對內核的深刻理解。

一 調試前的準備

在調試一個bug之前，我們所要做的準備工作有：

- 有一個被確認的bug。
- 包含這個bug的內核版本號，需要分析出這個bug在哪一個版本被引入，這個對於解決問題有極大的幫助。可以採用二分查找法來逐步鎖定bug引入版本號。
- 對內核代碼理解越深刻越好，同時還需要一點點運氣。
- 該bug可以復現。如果能夠找到復現規律，那麼離找到問題的原因就不遠了。
- 最小化系統。把可能產生bug的因素逐一排除掉。

二 內核中的bug

內核中的bug也是多種多樣的。它們的產生有無數的原因，同時表象也變化多端。從隱藏在源代碼中的錯誤到展現在目擊者面前的bug，其發作往往是一系列連鎖反應的事件才可能出發的。雖然內核調試有一定的困難，但是通過你的努力和理解，說不定你會喜歡上這樣的挑戰。

三 內核調試配置選項

學習編寫驅動程序要構建安裝自己的內核（標準主線內核）。最重要的原因之一是：內核開發者已經建立了多項用於調試的功能。但是由於這些功能會造成額外的輸出，並導致能下降，因此發行版廠商通常會禁止發行版內核中的調試功能。

1 內核配置

為了實現內核調試，在內核配置上增加了幾項：

```
1       Kernel hacking  --->
2      [*]   Magic SysRq key
3      [*]   Kernel debugging
4      [*]   Debug slab memory allocations
5      [*]   Spinlock and rw-lock debugging: basic checks
6      [*]   Spinlock debugging: sleep-inside-spinlock checking
7                 [*]   Compile the kernel with debug info
8      Device Drivers  --->
9                 Generic Driver Options  --->
10                [*]   Driver Core verbose debug messages
11     General setup  --->
12                [*]   Configure standard kernel features (for small systems)  --->
13                [*]   Load all symbols for debugging/ksymoops
```

啟用選項例如：

```
1     slab layer debugging（slab層調試選項）
2     high-memory debugging（高端內存調試選項）
3     I/O mapping debugging（I/O映射調試選項）
4     spin-lock debugging（自旋鎖調試選項）
5     stack-overflow checking（棧溢出檢查選項）
6     sleep-inside-spinlock checking（自旋鎖內睡眠選項）
```

2 調試原子操作

從內核2.5開發，為了檢查各類由原子操作引發的問題，內核提供了極佳的工具。

內核提供了一個原子操作計數器，它可以配置成，一旦在原子操作過程中，進城進入睡眠或者做了一些可能引起睡眠的操作，就打印警告信息並提供追蹤線索。

所以，包括在使用鎖的時候調用schedule()，正使用鎖的時候以阻塞方式請求分配內存等，各種潛在的bug都能夠被探測到。

下面這些選項可以最大限度地利用該特性：

```
1    CONFIG_PREEMPT = y
2    CONFIG_DEBUG_KERNEL = y
3    CONFIG_KLLSYMS = y
4    CONFIG_SPINLOCK_SLEEP = y
```

四 引發bug並打印信息

1 BUG()和BUG_ON()

一些內核調用可以用來方便標記bug，提供斷言並輸出信息。最常用的兩個是BUG()和BUG_ON()。

定義在<include/asm-generic>中：

```
1    #ifndef HAVE_ARCH_BUG
2    #define BUG() do {
3       printk("BUG: failure at %s:%d/%s()! ", __FILE__, __LINE__, __FUNCTION__);
4       panic("BUG!");   /* 引發更嚴重的錯誤，不但打印錯誤消息，而且整個系統業會掛起 */
5    } while (0)
6    #endif
7
8    #ifndef HAVE_ARCH_BUG_ON
```

```
9         #define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while(0)
10     #endif
```

當調用這兩個宏的時候，它們會引發OOPS，導致棧的回溯和錯誤消息的打印。

※ 可以把這兩個調用當作斷言使用，如：BUG_ON(bad_thing);

2 dump_stack()

有些時候，只需要在終端上打印一下棧的回溯信息來幫助你調試。這時可以使用dump_stack()。這個函數只在終端上打印寄存器上下文和函數的跟蹤線索。

```
1        if (!debug_check) {
2            printk(KERN_DEBUG 「provide some information…/n」);
3            dump_stack();
4        }
```

五 printk()

內核提供的格式化打印函數。

1 printk函數的健壯性

健壯性是printk最容易被接受的一個特質，幾乎在任何地方，任何時候內核都可以調用它（中斷上下文、進程上下文、持有鎖時、多處理器處理時等）。

2 printk函數脆弱之處

在系統啟動過程中，終端初始化之前，在某些地方是不能調用的。如果真的需要調試系統啟動過程最開始的地方，有以下方法可以使用：

- 使用串口調試，將調試信息輸出到其他終端設備。
- 使用early_printk()，該函數在系統啟動初期就有打印能力。但它只支持部分硬件體系。

3 LOG等級

printk和printf一個主要的區別就是前者可以指定一個LOG等級。內核根據這個等級來判斷是否在終端上打印消息。內核把比指定等級高的所有消息顯示在終端。

可以使用下面的方式指定一個LOG級別：

printk(KERN_CRIT 「Hello, world!\n」);

注意，第一個參數並不一個真正的參數，因為其中沒有用於分隔級別（KERN_CRIT）和格式字符的逗號（,）。KERN_CRIT本身只是一個普通的字符串（事實上，它表示的是字符串 "<2>"；表 1 列出了完整的日誌級別清單）。作為預處理程序的一部分，C 會自動地使用一個名為 字符串串聯 的功能將這兩個字符串組合在一起。組合的結果是將日誌級別和用戶指定的格式字符串包含在一個字符串中。

內核使用這個指定LOG級別與當前終端LOG等級console_loglevel來決定是不是向終端打印。

下面是可使用的LOG等級：

```
1    #define KERN_EMERG      "<0>"   /* system is unusable                           */
2    #define KERN_ALERT        "<1>"   /* action must be taken immediately     */
3    #define KERN_CRIT           "<2>"   /* critical conditions
4    */
5    #define KERN_ERR            "<3>"   /* error conditions
6    */
7    #define KERN_WARNING  "<4>"   /* warning conditions                              */
8    #define KERN_NOTICE       "<5>"   /* normal but significant condition         */
9    #define KERN_INFO            "<6>"   /* informational
     */
     #define KERN_DEBUG        "<7>"   /* debug-level messages                        */
     #define KERN_DEFAULT     "<d>"   /* Use the default kernel loglevel           */
```

注意，如果調用者未將日誌級別提供給 printk，那麼系統就會使用默認值 KERN_WARNING "<4>"（表示只有 KERN_WARNING 級別以上的日誌消息會被記錄）。由於默認值存在變化，所以在使用時最好指定LOG級別。有LOG級別的一個好處就是我們可以選擇性的輸出LOG。比如平時我們只需要打印KERN_WARNING級別以上的關鍵性

LOG，但是調試的時候，我們可以選擇打印KERN_DEBUG等以上的詳細LOG。而這些都不需要我們修改代碼，只需要通過命令修改默認日誌輸出級別：

```
1     mtj@ubuntu :~$ cat /proc/sys/kernel/printk
2     4 4 1 7
3     mtj@ubuntu :~$ cat /proc/sys/kernel/printk_delay
4     0
5     mtj@ubuntu :~$ cat /proc/sys/kernel/printk_ratelimit
6     5
7     mtj@ubuntu :~$ cat /proc/sys/kernel/printk_ratelimit_burst
8     10
```

第一項定義了 printk API 當前使用的日誌級別。這些日誌級別表示了控制台的日誌級別、默認消息日誌級別、最小控制台日誌級別和默認控制台日誌級別。printk_delay 值表示的是 printk 消息之間的延遲毫秒數（用於提高某些場景的可讀性）。注意，這裡它的值為 0，而它是不可以通過 /proc 設置的。printk_ratelimit 定義了消息之間允許的最小時間間隔（當前定義為每 5 秒內的某個內核消息數）。消息數量是由 printk_ratelimit_burst 定義的（當前定義為 10）。如果您擁有一個非正式內核而又使用有帶寬限制的控制台設備（如通過串口）， 那麼這非常有用。注意，在內核中，速度限制是由調用者控制的，而不是在printk 中實現的。如果一個 printk 用戶要求進行速度限制，那麼該用戶就需要調用printk_ratelimit 函數。

4 記錄緩衝區

內核消息都被保存在一個LOG_BUF_LEN大小的環形隊列中。

關於LOG_BUF_LEN定義：

```
1      #define __LOG_BUF_LEN (1 << CONFIG_LOG_BUF_SHIFT)
```

※ 變量CONFIG_LOG_BUF_SHIFT在內核編譯時由配置文件定義，對於i386平台，其值定義如下（在linux26/arch/i386/defconfig中）：

```
1     CONFIG_LOG_BUF_SHIFT=18
```

記錄緩衝區操作：

① 消息被讀出到用戶空間時，此消息就會從環形隊列中刪除。

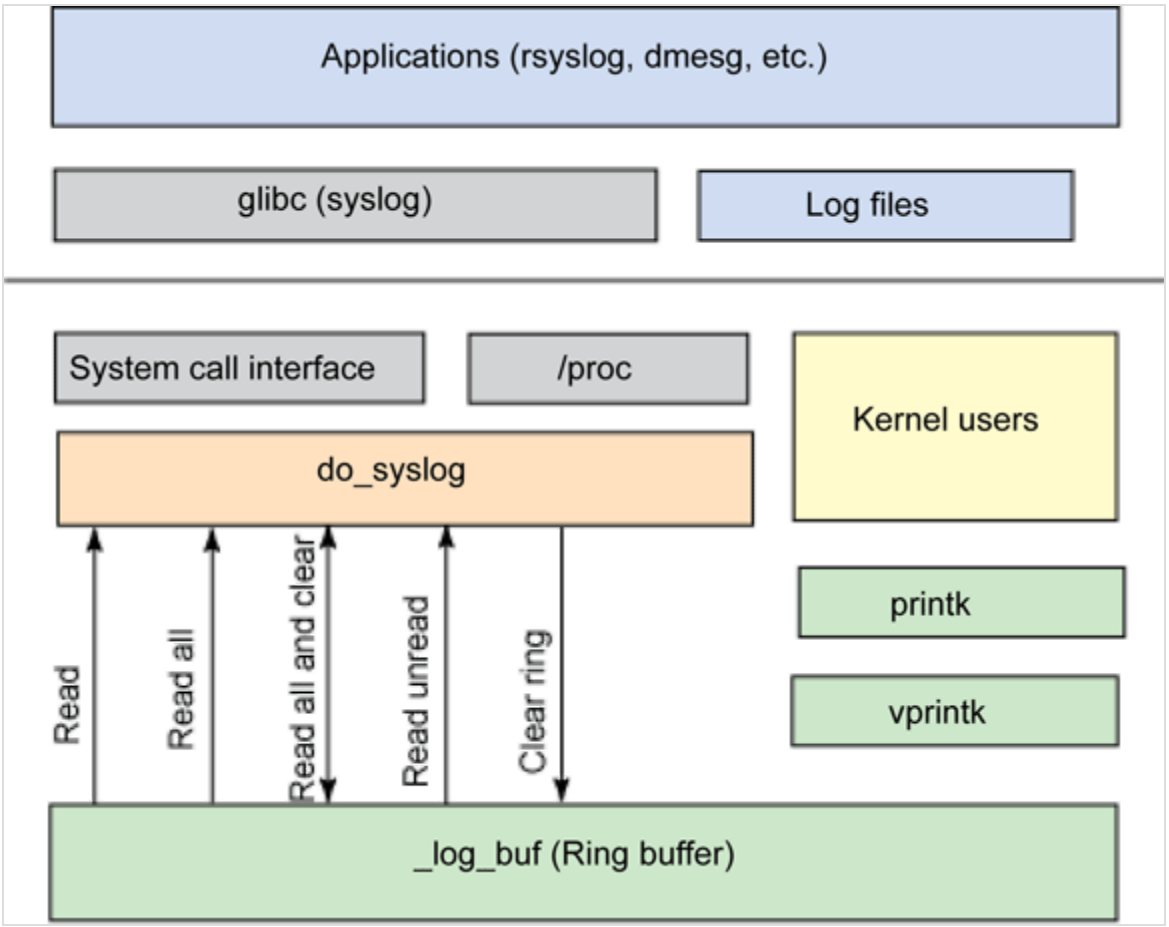
② 當消息緩衝區滿時，如果再有printk()調用時，新消息將覆蓋隊列中的老消息。

③ 在讀寫環形隊列時，同步問題很容易得到解決。

※ 這個紀錄緩衝區之所以稱為環形，是因為它的讀寫都是按照環形隊列的方式進行操作的。

5 syslogd/klogd

在標準的Linux系統上，用戶空間的守護進程klogd從紀錄緩衝區中獲取內核消息，再通過syslogd守護進程把這些消息保存在系統日誌文件中。klogd進程既可以從/proc/kmsg文件中，也可以通過syslog()系統調用讀取這些消息。默認情況下，它選擇讀取/proc方式實現。klogd守護進程在消息緩衝區有新的消息之前，一直處於阻塞狀態。一旦有新的內核消息，klogd被喚醒，讀出內核消息並進行處理。默認情況下，處理例程就是把內核消息傳給syslogd守護進程。syslogd守護進程一般把接收到的消息寫入/var/log/messages文件中。不過，還是可以通過/etc/syslog.conf文件來進行配置，可以選擇其他的輸出文件。



6 dmesg

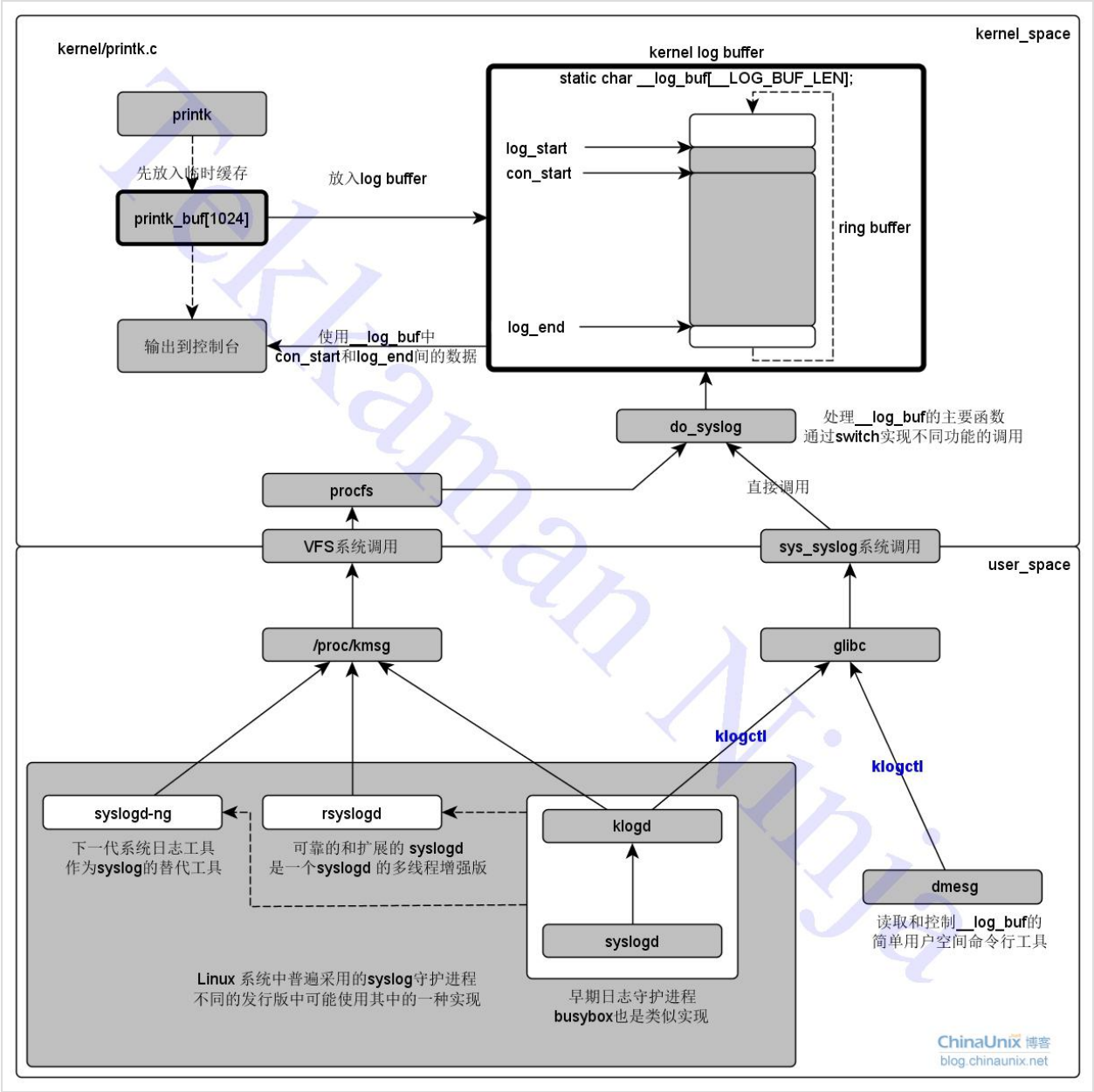
dmesg 命令也可用於打印和控制內核環緩衝區。這個命令使用 klogctl 系統調用來讀取內核環緩衝區，並將它轉發到標準輸出（stdout）。這個命令也可以用來清除內核環緩衝區（使用 -c 選項），設置控制台日誌級別（-n 選項），以及定義用於讀取內核日誌消息的緩衝區大小（-s 選項）。注意，如果沒有指定緩衝區大小，那麼 dmesg 會使用 klogctl 的SYSLOG_ACTION_SIZE_BUFFER 操作確定緩衝區大小。

7 注意

a) 雖然printk很健壯，但是看了源碼你就知道，這個函數的效率很低：做字符拷貝時一次只拷貝一個字節，且去調用console輸出可能還產生中斷。所以如果你的驅動在功能調試完成以後做性能測試或者發佈的時候千萬記得儘量減少printk輸出，做到僅在出錯時輸出少量信息。否則往console輸出無用信息影響性能。

b) printk的臨時緩存printk_buf只有1K，所有一次printk函數只能記錄<1K的信息到log buffer，並且printk使用的「ringbuffer」.

8 內核printk和日誌系統的總體結構



9 動態調試

動態調試是通過動態的開啟和禁止某些內核代碼來獲取額外的內核信息。

首先內核選項CONFIG_DYNAMIC_DEBUG應該被設置。所有通過pr_debug()/dev_debug()打印的信息都可以動態的顯示或不顯示。

可以通過簡單的查詢語句來篩選需要顯示的信息。

－源文件名

－函數名

－行號（包括指定範圍的行號）

－模塊名

－格式化字符串

將要打印信息的格式寫入<debugfs>/dynamic_debug/control中。

```
1    nullarbor:~ # echo 'file svcsock.c line 1603 +p' >      <debugfs>/dynamic_debug/control
```

參考：

1 內核日誌及printk結構淺析 -- Tekkaman Ninja

2 內核日誌：API 及實現

3 printk實現分析

4 dynamic-debug-howto.txt

六 內存調試工具

1 MEMWATCH

MEMWATCH 由 Johan Lindh 編寫，是一個開放源代碼 C 語言內存錯誤檢測工具，您可以自己下載它。只要在代碼中添加一個頭文件並在 gcc 語句中定義了 MEMWATCH 之後，您就可以跟蹤程序中的內存洩漏和錯誤了。MEMWATCH 支持ANSIC，它提供結果日誌紀錄，能檢測雙重釋放（double-free）、錯誤釋放（erroneous free）、沒有釋放的內存（unfreedmemory）、溢出和下溢等等。

清單 1. 內存樣本（test1.c）

```
1      #include <stdlib.h>
2      #include <stdio.h>
3      #include "memwatch.h"
4      int main(void)
5      {
6       char *ptr1;
7       char *ptr2;
8       ptr1 = malloc(512);
9       ptr2 = malloc(512);
10      ptr2 = ptr1;
11      free(ptr2);
12      free(ptr1);
13     }
```

清單 1 中的代碼將分配兩個 512 字節的內存塊，然後指向第一個內存塊的指針被設定為指向第二個內存塊。結果，第二個內存塊的地址丟失，從而產生了內存洩漏。

現在我們編譯清單 1 的 memwatch.c。下面是一個 makefile 示例：

test1

```
1     gcc -DMEMWATCH -DMW_STDIO test1.c memwatch
2     c -o test1
```

當您運行 test1 程序後，它會生成一個關於洩漏的內存的報告。清單 2 展示了示例 memwatch.log 輸出文件。

清單 2. test1 memwatch.log 文件

```
1      MEMWATCH 2.67 Copyright (C) 1992-1999 Johan Lindh
2      ...
3      double-free: <4> test1.c(15), 0x80517b4 was freed from test1.c(14)
4      ...
5      unfreed: <2> test1.c(11), 512 bytes at 0x80519e4
6      {FE FE FE FE FE FE FE FE FE FE FE FE ..............}
```

```
7      Memory usage statistics (global):
8       N)umber of allocations made: 2
9       L)argest memory usage : 1024
10      T)otal of all alloc() calls: 1024
11      U)nfreed bytes totals : 512
```

MEMWATCH 為您顯示真正導致問題的行。如果您釋放一個已經釋放過的指針，它會告訴您。對於沒有釋放的內存也一樣。日誌結尾部分顯示統計信息，包括洩漏了多少內存，使用了多少內存，以及總共分配了多少內存。

2 YAMD

YAMD 軟件包由 Nate Eldredge 編寫，可以查找 C 和 C++ 中動態的、與內存分配有關的問題。在撰寫本文時，YAMD 的最新版本為 0.32。請下載 yamd-0.32.tar.gz。執行 make 命令來構建程序；然後執行 make install 命令安裝程序並設置工具。

一旦您下載了 YAMD 之後，請在 test1.c 上使用它。請刪除 #include memwatch.h 並對 makefile 進行如下小小的修改：

使用 YAMD 的 test1

```
1     gcc -g test1.c -o test1
```

清單 3 展示了來自 test1 上的 YAMD 的輸出。

清單 3. 使用 YAMD 的 test1 輸出

```
1      YAMD version 0.32
2      Executable: /usr/src/test/yamd-0.32/test1
3      ...
4      INFO: Normal allocation of this block
5      Address 0x40025e00, size 512
6      ...
7      INFO: Normal allocation of this block
```

```
8      Address 0x40028e00, size 512
9      ...
10     INFO: Normal deallocation of this block
11     Address 0x40025e00, size 512
12     ...
13     ERROR: Multiple freeing At
14     free of pointer already freed
15     Address 0x40025e00, size 512
16     ...
17     WARNING: Memory leak
18     Address 0x40028e00, size 512
19     WARNING: Total memory leaks:
20     1 unfreed allocations totaling 512 bytes
21     *** Finished at Tue ... 10:07:15 2002
22     Allocated a grand total of 1024 bytes 2 allocations
23     Average of 512 bytes per allocation
24     Max bytes allocated at one time: 1024
25     24 K alloced internally / 12 K mapped now / 8 K max
26     Virtual program size is 1416 K
27     End.
```

YAMD 顯示我們已經釋放了內存，而且存在內存洩漏。讓我們在清單 4 中另一個樣本程序上試試 YAMD。

清單 4. 內存代碼（test2.c）

```
1      #include <stdlib.h>
2      #include <stdio.h>
3      int main(void)
4      {
5       char *ptr1;
6       char *ptr2;
7       char *chptr;
8       int i = 1;
```

```
9       ptr1 = malloc(512);
10      ptr2 = malloc(512);
11      chptr = (char *)malloc(512);
12      for (i; i <= 512; i++) {
13        chptr[i] = 'S';
14      }
15      ptr2 = ptr1;
16      free(ptr2);
17      free(ptr1);
18      free(chptr);
19     }
```

您可以使用下面的命令來啟動 YAMD：

```
1     ./run-yamd /usr/src/test/test2/test2
```

清單 5 顯示了在樣本程序 test2 上使用 YAMD 得到的輸出。YAMD 告訴我們在 for 循環中有「越界（out-of-bounds）」的情況。

清單 5. 使用 YAMD 的 test2 輸出

```
1      Running /usr/src/test/test2/test2
2      Temp output to /tmp/yamd-out.1243
3      *********
4      ./run-yamd: line 101: 1248 Segmentation fault (core dumped)
5      YAMD version 0.32
6      Starting run: /usr/src/test/test2/test2
7      Executable: /usr/src/test/test2/test2
8      Virtual program size is 1380 K
9      ...
10     INFO: Normal allocation of this block
11     Address 0x40025e00, size 512
12     ...
13     INFO: Normal allocation of this block
```

```
14     Address 0x40028e00, size 512
15     ...
16     INFO: Normal allocation of this block
17     Address 0x4002be00, size 512
18     ERROR: Crash
19     ...
20     Tried to write address 0x4002c000
21     Seems to be part of this block:
22     Address 0x4002be00, size 512
23     ...
24     Address in question is at offset 512 (out of bounds)
25     Will dump core after checking heap.
26     Done.
```

MEMWATCH 和 YAMD 都是很有用的調試工具，它們的使用方法有所不同。對於 MEMWATCH，您需要添加包含文件memwatch.h 並打開兩個編譯時間標記。對於鏈接（link）語句，YAMD 只需要 -g 選項。

3 Electric Fence

多數 Linux 分發版包含一個 Electric Fence 包，不過您也可以選擇下載它。Electric Fence 是一個由 Bruce Perens 編寫的malloc()調試庫。它就在您分配內存後分配受保護的內存。如果存在 fencepost 錯誤（超過數組末尾運行），程序就會產生保護錯誤，並立即結束。通過結合 Electric Fence 和 gdb，您可以精確地跟蹤到哪一行試圖訪問受保護內存。ElectricFence 的另一個功能就是能夠檢測內存洩漏。

七 strace

strace 命令是一種強大的工具，它能夠顯示所有由用戶空間程序發出的系統調用。strace 顯示這些調用的參數並返回符號形式的值。strace 從內核接收信息，而且不需要以任何特殊的方式來構建內核。將跟蹤信息發送到應用程序及內核開發者都很有用。在清單 6 中，分區的一種格式有錯誤，清單顯示了 strace 的開頭部分，內容是關於調出創建文件系統操作（mkfs ）的。strace 確定哪個調用導致問題出現。

清單 6. mkfs 上 strace 的開頭部分

```
1     execve("/sbin/mkfs.jfs", ["mkfs.jfs", "-f", "/dev/test1"], &
2     ...
3     open("/dev/test1", O_RDWR|O_LARGEFILE) = 4
4     stat64("/dev/test1", {st_mode=&, st_rdev=makedev(63, 255), ...}) = 0
5     ioctl(4, 0x40041271, 0xbfffe128) = -1 EINVAL (Invalid argument)
6     write(2, "mkfs.jfs: warning - cannot setb" ..., 98mkfs.jfs: warning -
7     cannot set blocksize on block device /dev/test1: Invalid argument )
8      = 98
9     stat64("/dev/test1", {st_mode=&, st_rdev=makedev(63, 255), ...}) = 0
10    open("/dev/test1", O_RDONLY|O_LARGEFILE) = 5
11    ioctl(5, 0x80041272, 0xbfffe124) = -1 EINVAL (Invalid argument)
12    write(2, "mkfs.jfs: can\'t determine device"..., ..._exit(1)
13     = ?
```

清單 6 顯示 ioctl 調用導致用來格式化分區的 mkfs 程序失敗。 ioctl BLKGETSIZE64 失敗。（ BLKGET-SIZE64 在調用 ioctl的源代碼中定義。）BLKGETSIZE64 ioctl 將被添加到 Linux 中所有的設備，而在這裡，邏輯卷管理器還不支持它。因此，如果BLKGETSIZE64 ioctl 調用失敗，mkfs 代碼將改為調用較早的 ioctl 調用；這使得 mkfs 適用於邏輯卷管理器。

參考：

http://www.ibm.com/developerworks/cn/linux/sdk/l-debug/index.html#resources

八 OOPS

OOPS（也稱 Panic）消息包含系統錯誤的細節，如 CPU 寄存器的內容等。是內核告知用戶有不幸發生的最常用的方式。

內核只能發佈OOPS，這個過程包括向終端上輸出錯誤消息，輸出寄存器保存的信息，並輸出可供跟蹤的回溯線索。通常，發送完OOPS之後，內核會處於一種不穩定的狀態。

OOPS的產生有很多可能原因，其中包括內存訪問越界或非法的指令等。

※ 作為內核的開發者，必定將會經常處理OOPS。

※ OOPS中包含的重要信息，對所有體系結構的機器都是完全相同的：寄存器上下文和回溯線索（回溯線索顯示了導致錯誤發生的函數調用鏈）。

1 ksymoops

在 Linux 中，調試系統崩潰的傳統方法是分析在發生崩潰時發送到系統控制台的 Oops 消息。一旦您掌握了細節，就可以將消息發送到 ksymoops 實用程序，它將試圖將代碼轉換為指令並將堆棧值映射到內核符號。

※ 如：回溯線索中的地址，會通過ksymoops轉化成名稱可見的函數名。

ksymoops需要幾項內容：Oops 消息輸出、來自正在運行的內核的 System.map 文件，還有 /proc/ksyms、vmlinux和/proc/modules。

關於如何使用 ksymoops，內核源代碼 /usr/src/linux/Documentation/oops-tracing.txt 中或 ksymoops 手冊頁上有完整的說明可以參考。Ksymoops 反彙編代碼部分，指出發生錯誤的指令，並顯示一個跟蹤部分表明代碼如何被調用。

首先，將 Oops 消息保存在一個文件中以便通過 ksymoops 實用程序運行它。清單 7 顯示了由安裝 JFS 文件系統的 mount命令創建的 Oops 消息。

清單 7. ksymoops 處理後的 Oops 消息

```
1       ksymoops 2.4.0 on i686 2.4.17. Options used
2      ... 15:59:37 sfb1 kernel: Unable to handle kernel NULL pointer dereference at
3      virtual address 0000000
4      ... 15:59:37 sfb1 kernel: c01588fc
5      ... 15:59:37 sfb1 kernel: *pde = 0000000
6      ... 15:59:37 sfb1 kernel: Oops: 0000
7      ... 15:59:37 sfb1 kernel: CPU:    0
8      ... 15:59:37 sfb1 kernel: EIP:    0010:[jfs_mount+60/704]
```

```
9      ... 15:59:37 sfb1 kernel: Call Trace: [jfs_read_super+287/688]
10     [get_sb_bdev+563/736] [do_kern_mount+189/336] [do_add_mount+35/208]
11     [do_page_fault+0/1264]
12     ... 15:59:37 sfb1 kernel: Call Trace: [<c0155d4f>]...
13     ... 15:59:37 sfb1 kernel: [<c0106e04 ...
14     ... 15:59:37 sfb1 kernel: Code: 8b 2d 00 00 00 00 55 ...
15     >>EIP; c01588fc <jfs_mount+3c/2c0> <=====
16     ...
17     Trace; c0106cf3 <system_call+33/40>
18     Code; c01588fc <jfs_mount+3c/2c0>
19     00000000 <_EIP>:
20     Code; c01588fc <jfs_mount+3c/2c0>  <=====
21       0: 8b 2d 00 00 00 00 mov 0x0,%ebp    <=====
22     Code; c0158902 <jfs_mount+42/2c0>
23       6:  55 push %ebp
```

接下來，您要確定 jfs_mount 中的哪一行代碼引起了這個問題。Oops 消息告訴我們問題是由位於偏移地址 3c 的指令引起的。做這件事的辦法之一是對 jfs_mount.o 文件使用 objdump 實用程序，然後查看偏移地址 3c。Objdump 用來反彙編模塊函數，看看您的 C 源代碼會產生什麼彙編指令。清單 8 顯示了使用 objdump 後您將看到的內容，接著，我們查看jfs_mount 的 C 代碼，可以看到空值是第 109 行引起的。偏移地址 3c 之所以很重要，是因為 Oops 消息將該處標識為引起問題的位置。

清單 8. jfs_mount 的彙編程序清單

```
1      109 printk("%d\n",*ptr);
2      objdump jfs_mount.o
3      jfs_mount.o: file format elf32-i386
4      Disassembly of section .text:
5      00000000 <jfs_mount>:
6        0:55 push %ebp
7       ...
8       2c: e8 cf 03 00 00   call   400 <chkSuper>
9       31: 89 c3       mov     %eax,%ebx
10      33: 58     pop     %eax
```

```
11      34: 85 db       test %ebx,%ebx
12      36: 0f 85 55 02 00 00 jne 291 <jfs_mount+0x291>
13      3c: 8b 2d 00 00 00 00 mov 0x0,%ebp << problem line above
14      42: 55 push %ebp
```

2 kallsyms

開發版2.5內核引入了kallsyms特性，它可以通過定義CONFIG_KALLSYMS編譯選項啟用。該選項可以載入內核鏡像所對應的內存地址的符號名稱（即函數名），所以內核可以打印解碼之後的跟蹤線索。相應，解碼OOPS也不再需要System.map和ksymoops工具了。另外，

這樣做，會使內核變大些，因為地址對應符號名稱必須始終駐留在內核所在內存上。

```
1     #cat /proc/kallsyms
2     c0100240   T    _stext
3     c0100240   t    run_init_process
4     c0100240   T      stext
5     c0100269   t    init
6         …
```

3 Kdump

3.1 Kdump 的基本概念

3.1.1 什麼是 kexec ？

Kexec 是實現 kdump 機制的關鍵，它包括 2 個組成部分：一是內核空間的系統調用 kexec_load，負責在生產內核（production kernel 或 first kernel）啟動時將捕獲內核（capture kernel 或 sencond kernel）加載到指定地址。二是用戶空間的工具 kexec-tools，他將捕獲內核的地址傳遞給生產內核，從而在系統崩潰的時候能夠找到捕獲內核的地址並運行。沒有 kexec 就沒有 kdump。先有 kexec 實現了在一個內核中可以啟動另一個內核，才讓 kdump 有了用武之地。kexec 原來的目的是為了節省 kernel 開發人員重啟系統的時間，誰能想到這個「偷懶」的技術卻孕育了最成功的內存轉存機制呢？

3.1.2 什麼是 kdump ？

Kdump 的概念出現在 2005 左右，是迄今為止最可靠的內核轉存機制，已經被主要的 linux™ 廠商選用。kdump是一種先進的基於 kexec 的內核崩潰轉儲機制。當系統崩潰時，kdump 使用 kexec 啟動到第二個內核。第二個內核通常叫做捕獲內核，以很小內存啟動以捕獲轉儲鏡像。第一個內核保留了內存的一部分給第二內核啟動用。由於 kdump 利用 kexec 啟動捕獲內核，繞過了 BIOS，所以第一個內核的內存得以保留。這是內核崩潰轉儲的本質。

kdump 需要兩個不同目的的內核，生產內核和捕獲內核。生產內核是捕獲內核服務的對像。捕獲內核會在生產內核崩潰時啟動起來，與相應的 ramdisk 一起組建一個微環境，用以對生產內核下的內存進行收集和轉存。

3.1.3 如何使用 kdump

構建系統和 dump-capture 內核，此操作有 2 種方式可選：

1）構建一個單獨的自定義轉儲捕獲內核以捕獲內核轉儲；

2） 或者將系統內核本身作為轉儲捕獲內核，這就不需要構建一個單獨的轉儲捕獲內核。

方法（2）只能用於可支持可重定位內核的體系結構上；目前 i386，x86_64，ppc64 和 ia64 體系結構支持可重定位內核。構建一個可重定位內核使得不需要構建第二個內核就可以捕獲轉儲。但是可能有時想構建一個自定義轉儲捕獲內核以滿足特定要求。

3.1.4 如何訪問捕獲內存

在內核崩潰之前所有關於核心映像的必要信息都用 ELF 格式編碼並存儲在保留的內存區域中。ELF 頭所在的物理地址被作為命令行參數（fcorehdr=）傳遞給新啟動的轉儲內核。

在 i386 體系結構上，啟動的時候需要使用物理內存開始的 640K，而不管操作系統內核轉載在何處。因此，這個 640K 的區域在重新啟動第二個內核的時候由 kexec 備份。

在第二個內核中，「前一個系統的內存」可以通過兩種方式訪問：

1） 通過 /dev/oldmem 這個設備接口。

一個「捕捉」設備可以使用「raw」（裸的）方式 「讀」這個設備文件並寫出到文件。這是關於內存的 「裸」的數據轉儲，同時這些分析 / 捕捉工具應該足夠「智能」從而可以知道從哪裡可以得到正確的信息。ELF 文件頭（通過命令行參數傳遞過來的 elfcorehdr）可能會有幫助。

2） 通過 /proc/vmcore。

這個方式是將轉儲輸出為一個 ELF 格式的文件，並且可以使用一些文件拷貝命令（比如 cp，scp 等）將信息讀出來。同時，gdb 可以在得到的轉儲文件上做一些調試（有限的）。這種方式保證了內存中的頁面都以正確的途徑被保存（注意內存開始的 640K 被重新映射了)。

3.1.5 kdump 的優勢

1） 高可靠性

崩潰轉儲數據可從一個新啟動內核的上下文中獲取，而不是從已經崩潰內核的上下文。

2） 多版本支持

LKCD(Linux Kernel Crash Dump)，netdump，diskdump 已被納入 LDPs(Linux Documen-tation Project) 內核。SUSE 和 RedHat 都對 kdump 有技術支持。

3.2 Kdump 實現流程

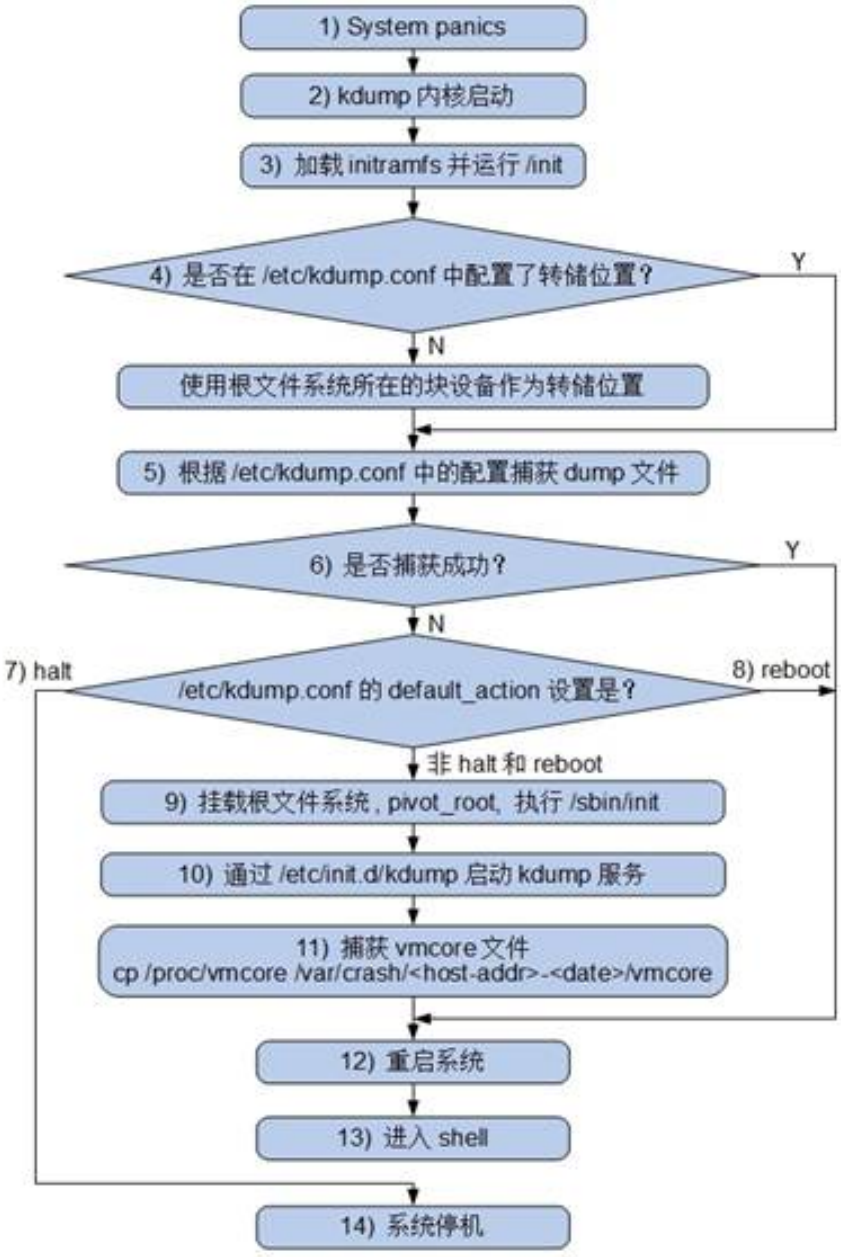

1) System panics
2) kdump 内核启动
3) 加载 initramfs 并运行 /init
4) 是否在 /etc/kdump.conf 中配置了转储位置？
Y
N
使用根文件系统所在的块设备作为转储位置
5) 根据 /etc/kdump.conf 中的配置捕获 dump 文件
6) 是否捕获成功？
Y
N
7) halt
/etc/kdump.conf 的 default_action 设置是？
8) reboot
非 halt 和 reboot
9) 挂载根文件系统，pivot_root，执行 /sbin/init
10) 通过 /etc/init.d/kdump 启动 kdump 服务
11) 捕获 vmcore 文件
cp /proc/vmcore /var/crash/<host-addr>-<date>/vmcore
12) 重启系统
13) 进入 shell
14) 系统停机

圖 1. RHEL6.2 執行流程

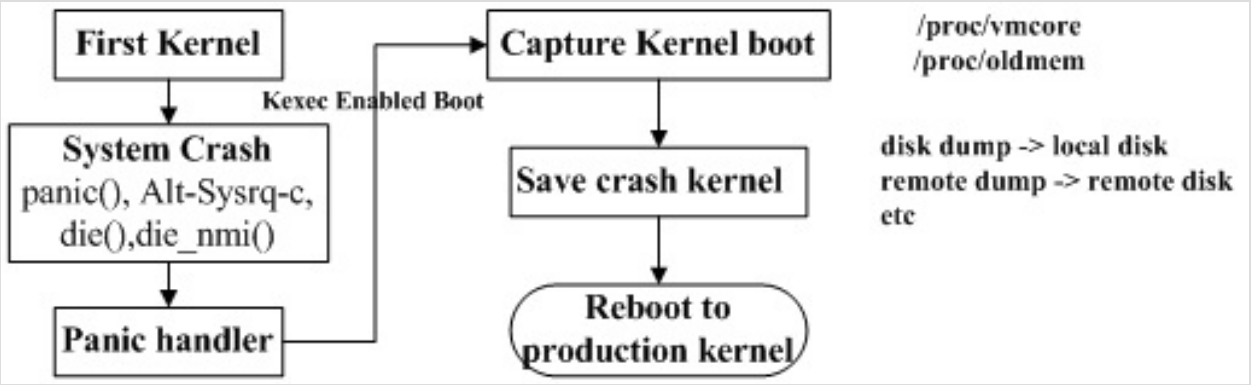


圖 2. sles11 執行流程

3.3 配置 kdump

3.3.1 安裝軟件包和實用程序

Kdump 用到的各種工具都在 kexec-tools 中。kernel-debuginfo 則是用來分析 vmcore 文件。從 rhel5 開始，kexec-tools 已被默認安裝在發行版。而 novell 也在 sles10 發行版中把 kdump 集成進來。所以如果使用的是 rhel5 和 sles10 之後的發行版，那就省去了安裝 kexec-tools 的步驟。而如果需要調試 kdump 生成的 vmcore文件，則需要手動安裝 kernel-debuginfo 包。檢查安裝包操作：

```
3.3.2 參數相關設置 uli13lp1:/ # rpm -qa|grep kexec
 kexec-tools-2.0.0-53.43.10
 uli13lp1:/ # rpm -qa 'kernel*debuginfo*'
 kernel-default-debuginfo-3.0.13-0.27.1
 kernel-ppc64-debuginfo-3.0.13-0.27.1
```

系統內核設置選項和轉儲捕獲內核配置選擇在《使用 Crash 工具分析 Linux dump 文件》一文中已有說明，在此不再贅述。僅列出內核引導參數設置以及配置文件設置。

1） 修改內核引導參數，為啟動捕獲內核預留內存

通過下面的方法來配置 kdump 使用的內存大小。添加啟動參數"crashkernel=Y@X"，這裡，Y 是為 kdump 捕捉內核保留的內存，X 是保留部分內存的開始位置。

- 對於 i386 和 x86_64, 編輯 /etc/grub.conf, 在內核行的最後添加"crashkernel=128M" 。
- 對於 ppc64，在 /etc/yaboot.conf 最後添加"crashkernel=128M"。

在 ia64, 編輯 /etc/elilo.conf，添加"crashkernel=256M"到內核行。

2） kdump 配置文件

kdump 的配置文件是 /etc/kdump.conf（RHEL6.2）；/etc/sysconfig/kdump(SLES11 sp2)。每個文件頭部都有選項說明，可以根據使用需求設置相應的選項。

3.3.3 啟動 kdump 服務

在設置了預留內存後，需要重啟機器，否則 kdump 是不可使用的。啟動 kdump 服務：

Rhel6.2：

```
# chkconfig kdump on
# service kdump status
 Kdump is operational
# service kdump start
```

SLES11SP2：

```
# chkconfig boot.kdump on
# service boot.kdump start
```

3.3.4 測試配置是否有效

可以通過 kexec 加載內核鏡像，讓系統準備好去捕獲一個崩潰時產生的 vmcore。可以通過 sysrq 強制系統崩潰。

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

這造成內核崩潰，如配置有效，系統將重啟進入 kdump 內核，當系統進程進入到啟動 kdump 服務的點時，vmcore 將會拷貝到你在 kdump 配置文件中設置的位置。RHEL 的缺省目錄是：/var/crash；SLES 的缺省目錄是：/var/log/dump。然後系統重啟進入到正常的內核。一旦回覆到正常的內核，就可以在上述的目錄下發現 vmcore

文件，即內存轉儲文件。可以使用之前安裝的 kernel-debuginfo 中的 crash 工具來進行分析（crash 的更多詳細用法將在本系列後面的文章中有介紹）。

```
# crash /usr/lib/debug/lib/modules/2.6.17-1.2621.el5/vmlinux
/var/crash/2006-08-23-15:34/vmcore
crash> bt
```

3.4 載入「轉儲捕獲」內核

需要引導系統內核時，可使用如下步驟和命令載入「轉儲捕獲」內核：

```
kexec -p <dump-capture-kernel> \
        --initrd=<initrd-for-dump-capture-kernel> --args-linux \
        --append="root=<root-dev> init 1 irqpoll"
```

裝載轉儲捕捉內核的注意事項：

- 轉儲捕捉內核應當是一個 vmlinux 格式的映像（即是一個未壓縮的 ELF 映像文件），而不能是 bzImage 格式；
- 默認情況下，ELF 文件頭採用 ELF64 格式存儲以支持那些擁有超過 4GB 內存的系統。但是可以指定「--elf32-core-headers」標誌以強制使用 ELF32 格式的 ELF 文件頭。這個標誌是有必要注意的，一個重要的原因就是：當前版本的 GDB 不能在一個 32 位系統上打開一個使用 ELF64 格式的 vmcore 文件。ELF32 格式的文件頭不能使用在一個「沒有物理地址擴展」（non-PAE）的系統上（即：少於 4GB 內存的系統）；
- 一個「irqpoll」的啟動參數可以減低由於在「轉儲捕獲內核」中使用了「共享中斷」技術而導致出現驅動初始化失敗這種情況發生的概率；
- 必須指定 <root-dev>，指定的格式是和要使用根設備的名字。具體可以查看 mount 命令的輸出；「init 1」這個命令將啟動「轉儲捕捉內核」到一個沒有網絡支持的單用戶模式。如果你希望有網絡支持，那麼使用「init 3」。

3.5 後記

Kdump 是一個強大的、靈活的內核轉儲機制，能夠在生產內核上下文中執行捕獲內核是非常有價值的。本文僅介紹在 RHEL6.2 和 SLES11 中如何配置 kdump。望拋磚引玉，對閱讀本文的讀者有益。

參考：

1 kallsyms的分析

2 深入探索 Kdump

九 KGDB

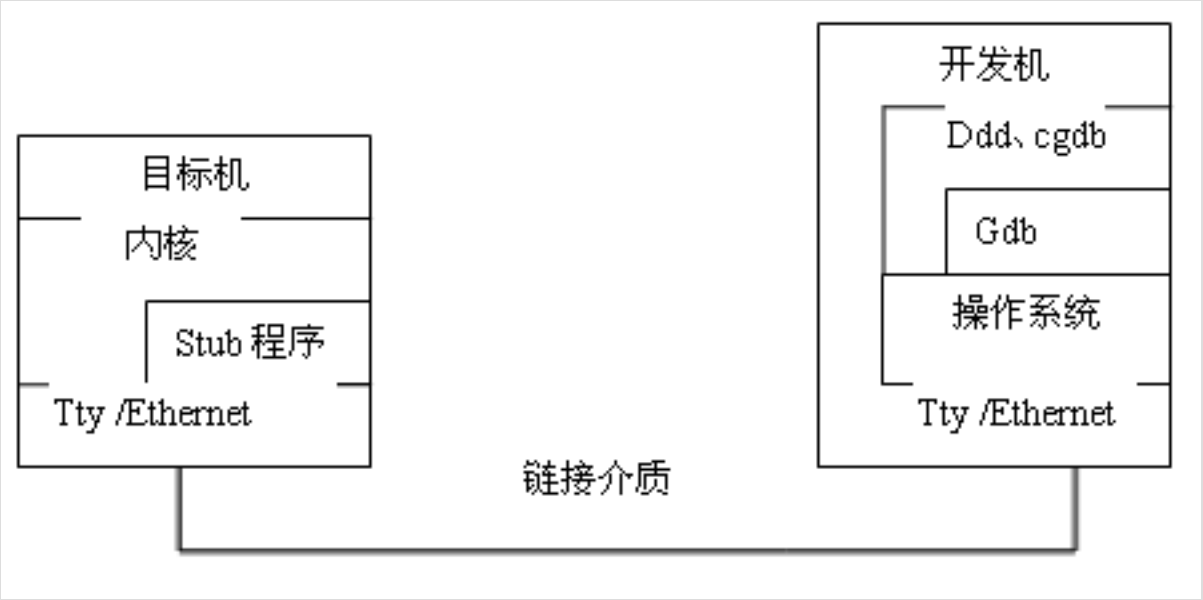
kgdb提供了一種使用 gdb調試 Linux 內核的機制。使用KGDB可以像調試普通的應用程序那樣，在內核中進行設置斷點、檢查變量值、單步跟蹤程序運行等操作。使用KGDB調試時需要兩台機器，一台作為開發機（Development Machine）,另一台作為目標機（Target Machine），兩台機器之間通過串口或者以太網口相連。串口連接線是一根RS-232接口的電纜，在其內部兩端的第2腳（TXD）與第3腳（RXD）交叉相連，第7腳（接地腳）直接相連。調試過程中，被調試的內核運行在目標機上，gdb調試器運行在開發機上。

目前，kgdb發佈支持i386、x86_64、32-bit PPC、SPARC等幾種體系結構的調試器。

1 kgdb的調試原理

安裝kgdb調試環境需要為Linux內核應用kgdb補丁，補丁實現的gdb遠程調試所需要的功能包括命令處理、陷阱處理及串口通訊3個主要的部分。kgdb補丁的主要作用是在Linux內核中添加了一個調試Stub。調試Stub是Linux內核中的一小段代碼，提供了運行gdb的開發機和所調試內核之間的一個媒介。gdb和調試stub之間通過gdb串行協議進行通訊。gdb串行協議是一種基於消息的ASCII碼協議，包含了各種調試命令。當設置斷點時，kgdb負責在設置斷點的指令前增加一條trap指令，當執行到斷點時控制權就轉移到調試stub中去。此時，調試stub的任務就是使用遠程串行通信協議將當前環境傳送給gdb，然後從gdb處接受命令。gdb命令告訴stub下一步該做什麼，當stub收到繼續執行的命

令時，將恢復程序的運行環境，把對CPU的控制權重新交還給內核



2 Kgdb的安装與設置

下面我們將以Linux 2.6.7內核為例詳細介紹kgdb調試環境的建立過程。

2.1 軟硬件準備

以下軟硬件配置取自筆者進行試驗的系統配置情況：

硬件	目标机	开发机	备注
IP 地址	192. 168. 5. 13	192. 168. 6. 13	—
连接端口	Com1	Com1	试验选用串口连接，调试端口的使用可选用以太网口
操作系统	Fedora 3	Fedora 3	选用 redhat 7.3 或以后版本
Linux 内核	Linux 2.6.7		—
kgdb 内核补丁	linux-2.6.7-kgdb-2.2.tar.tar		下载与 Linux 内核版本相对应的 kgdb 补丁。
串口线	使用空调制解调器电缆		或者按要求自己制作串口电缆

表 1：系统软硬件配置表

kgdb補丁的版本遵循如下命名模式：Linux-A-kgdb-B，其中A表示Linux的內核版本號，B為kgdb的版本號。以試驗使用的kgdb補丁為例，linux內核的版本為linux-2.6.7，補丁版本為kgdb-2.2。

物理連接好串口線後，使用以下命令來測試兩台機器之間串口連接情況，stty命令可以對串口參數進行設置：

在development機上執行：

```
1     stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

在target機上執行：

```
1     stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

在developement機上執行：

```
1     echo hello > /dev/ttyS0
```

在target機上執行：

```
1     cat /dev/ttyS0
```

如果串口連接沒問題的話在將在target機的屏幕上顯示"hello"。

2.2 安裝與配置

下面我們需要應用kgdb補丁到Linux內核，設置內核選項並編譯內核。這方面的資料相對較少，筆者這裡給出詳細的介紹。下面的工作在開發機（developement）上進行，以上面介紹的試驗環境為例，某些具體步驟在實際的環境中可能要做適當的改動：

I、內核的配置與編譯

```
1     [root@lisl tmp]# tar -jxvf linux-2.6.7.tar.bz2
2     [root@lisl tmp]#tar -jxvf linux-2.6.7-kgdb-2.2.tar.tar
3     [root@lisl tmp]#cd inux-2.6.7
```

請參照目錄補丁包中文件README給出的説明，執行對應體系結構的補丁程序。由於試驗在i386體系結構上完成，所以只需要安裝一下補丁：core-lite.patch、i386-lite.patch、8250.patch、eth.patch、core.patch、i386.patch。應用補丁文件時，請遵循kgdb軟件包內series文件所指定的順序，否則可能會帶來預想不到的問題。eth.patch文件是選擇以太網口作為調試的連接端口時需要運用的補丁。

應用補丁的命令如下所示：

```
1    [root@lisl tmp]#patch -p1 <../linux-2.6.7-kgdb-2.2/core-lite.patch
```

如果內核正確，那麼應用補丁時應該不會出現任何問題（不會產生*.rej文件）。為Linux內核添加了補丁之後，需要進行內核的配置。內核的配置可以按照你的習慣選擇配置Linux內核的任意一種方式。

```
1    [root@lisl tmp]#make menuconfig
```

在內核配置菜單的Kernel hacking選項中選擇kgdb調試項，例如：

```
1    [*] KGDB: kernel debugging with remote gdb
2          Method for KGDB communication (KGDB: On generic serial port (8250))  --->
3     [*] KGDB: Thread analysis
4     [*] KGDB: Console messages through gdb
5    [root@lisl tmp]#make
```

編譯內核之前請注意Linux目錄下Makefile中的優化選項，默認的Linux內核的編譯都以-O2的優化級別進行。在這個優化級別之下，編譯器要對內核中的某些代碼的執行順序進行改動，所以在調試時會出現程序運行與代碼順序不一致的情況。可以把Makefile中的-O2選項改為-O,但不可去掉-O，否則編譯會出問題。為了使編譯後的內核帶有調試信息，注意在編譯內核的時候需要加上-g選項。

不過，當選擇"Kernel debugging->Compile the kernel with debug info"選項後配置系統將自動打開調試選項。另外，選擇"kernel debugging with remote gdb"後，配置系統將自動打開"Compile the kernel with debug info"選項。

內核編譯完成後，使用scp命令進行將相關文件拷貝到target機上(當然也可以使用其它的網絡工具，如rcp)。

```
1    [root@lisl tmp]#scp arch/i386/boot/bzImage root@192.168.6.13:/boot/vmlinuz-2.6.7-kgdb
2    [root@lisl tmp]#scp System.map root@192.168.6.13:/boot/System.map-2.6.7-kgdb
```

如果系統啟動使所需要的某些設備驅動沒有編譯進內核的情況下，那麼還需要執行如下操作：

```
1     [root@lisl tmp]#mkinitrd /boot/initrd-2.6.7-kgdb 2.6.7
2     [root@lisl tmp]#scp initrd-2.6.7-kgdb root@192.168.6.13:/boot/ initrd-2.6.7-kgdb
```

II、kgdb的啟動

在將編譯出的內核拷貝的到target機器之後，需要配置系統引導程序，加入內核的啟動選項。以下是kgdb內核引導參數的說明：

kgdboe=@local-ip/,@remote-ip/	当使用网络接口作为调试作为启动参数告诉 stub，指定 target 机和 develop 机的 IP 地址。
2.0 版本以前的 kgdb： gdb gdbttyS=1 gdbbaud=115200	
启动参数	含义
Gdb	内核启动时等待 gdb 连接
gdbttyS	该选项指定 kgdb stub 使用哪一个串口进行通讯。取值为 0－3 分别代表 ttyS0 到 ttyS3 端口。
Gdbbaud	指定串口的波特率，波特率范围为 9600 to 115200
2.0 版本以后的 kgdb： kgdbwait kgdb8250=0,115200	
Kgdbwait	内核启动时等待 gdb 连接
kgdb8250=<port number>,<port speed>	该选项指定 kgdb stub 使用哪一个串口进行通讯，取值为 0－3。并指定端口的波特率，所支持的波特率为 9600, 19200, 38400, 57600 and 115200。

表 2：kgdb 内核引导参数

如表中所述，在kgdb 2.0版本之後內核的引導參數已經與以前的版本有所不同。使用grub引導程序時，直接將kgdb參數作為內核vmlinuz的引導參數。下面給出引導器的配置示例。

```
1     title 2.6.7 kgdb
2     root (hd0,0)
3     kernel /boot/vmlinuz-2.6.7-kgdb ro root=/dev/hda1 kgdbwait kgdb8250=1,115200
```

在使用lilo作為引導程序時，需要把kgdb參放在由append修飾的語句中。下面給出使用lilo作為引導器時的配置示例。

```
image=/boot/vmlinuz-2.6.7-kgdb
label=kgdb
   read-only
   root=/dev/hda3
append="gdb gdbttyS=1 gdbbaud=115200"
```

保存好以上配置後重新啟動計算機，選擇啟動帶調試信息的內核，內核將在短暫的運行後在創建init內核線程之前停下來，打印出以下信息，並等待開發機的連接。

Waiting for connection from remote gdb...

在開發機上執行：

```
gdb
file vmlinux
set remotebaud 115200
target remote /dev/ttyS0
```

其中vmlinux是指向源代碼目錄下編譯出來的Linux內核文件的鏈接，它是沒有經過壓縮的內核文件，gdb程序從該文件中得到各種符號地址信息。

這樣，就與目標機上的kgdb調試接口建立了聯繫。一旦建立聯接之後，對Linux內的調試工作與對普通的運用程序的調試就沒有什麼區別了。任何時候都可以通過鍵入ctrl+c打斷目標機的執行，進行具體的調試工作。

在kgdb 2.0之前的版本中，編譯內核後在arch/i386/kernel目錄下還會生成可執行文件gdbstart。將該文件拷貝到target機器的/boot目錄下，此時無需更改內核的啟動配置文件，直接使用命令：

```
[root@lisl boot]#gdbstart -s 115200 -t /dev/ttyS0
```

可以在KGDB內核引導啟動完成後建立開發機與目標機之間的調試聯繫。

2.3 通過網絡接口進行調試

kgdb也支持使用以太網接口作為調試器的連接端口。在對Linux內核應用補丁包時，需應用eth.patch補丁文件。配置內核時在Kernel hacking中選擇kgdb調試項，配置kgdb調試端口為以太網接口，例如：

```
[*]KGDB: kernel debugging with remote gdb
Method for KGDB communication (KGDB: On ethernet)  --->
( ) KGDB: On generic serial port (8250)
(X) KGDB: On ethernet
```

另外使用eth0網口作為調試端口時，grub.list的配置如下：

```
1    title 2.6.7 kgdb
2    root (hd0,0)
3    kernel /boot/vmlinuz-2.6.7-kgdb ro root=/dev/hda1 kgdbwait
     kgdboe=@192.168.5.13/,@192.168. 6.13/
```

其他的過程與使用串口作為連接端口時的設置過程相同。

注意：儘管可以使用以太網口作為kgdb的調試端口，使用串口作為連接端口更加簡單易行，kgdb項目組推薦使用串口作為調試端口。

2.4 模塊的調試方法

內核可加載模塊的調試具有其特殊性。由於內核模塊中各段的地址是在模塊加載進內核的時候才最終確定的，所以develop機的gdb無法得到各種符號地址信息。所以，使用kgdb調試模塊所需要解決的一個問題是，需要通過某種方法獲得可加載模塊的最終加載地址信息，並把這些信息加入到gdb環境中。

I、在Linux 2.4內核中的內核模塊調試方法

在Linux2.4.x內核中，可以使用insmod -m命令輸出模塊的加載信息，例如：

```
1    [root@lisl tmp]# insmod -m hello.ko >modaddr
```

查看模塊加載信息文件modaddr如下：

```
1    .this           00000060  c88d8000  2**2
2    .text           00000035  c88d8060  2**2
3    .rodata         00000069  c88d80a0  2**5
4    ……
5    .data           00000000  c88d833c  2**2
6    .bss            00000000  c88d833c  2**2
7    ……
```

在這些信息中，我們關心的只有4個段的地址:.text、.rodata、.data、.bss。在development機上將以上地址信息加入到gdb中,這樣就可以進行模塊功能的測試了。

```
1    (gdb) Add-symbol-file hello.o 0xc88d8060 -s .data 0xc88d80a0 -s .rodata 0xc88d80a0 -s
     .bss 0x c88d833c
```

這種方法也存在一定的不足，它不能調試模塊初始化的代碼，因為此時模塊初始化代碼已經執行過了。而如果不執行模塊的加載又無法獲得模塊插入地址，更不可能在模塊初始化之前設置斷點了。對於這種調試要求可以採用以下替代方法。

在target機上用上述方法得到模塊加載的地址信息，然後再用rmmod卸載模塊。在development機上將得到的模塊地址信息導入到gdb環境中，在內核代碼的調用初始化代碼之前設置斷點。這樣，在target機上再次插入模塊時，代碼將在執行模塊初始化之前停下來，這樣就可以使用gdb命令調試模塊初始化代碼了。

另外一種調試模塊初始化函數的方法是：當插入內核模塊時，內核模塊機制將調用函數sys_init_module(kernel/modle.c)執行對內核模塊的初始化，該函數將調用所插入模塊的初始化函數。程序代碼片斷如下：

```
1     …… ……
2     if (mod->init != NULL)
3     ret = mod->init();
4     …… ……
```

在該語句上設置斷點，也能在執行模塊初始化之前停下來。

II、在Linux 2.6.x內核中的內核模塊調試方法

Linux 2.6之後的內核中，由於module-init-tools工具的更改，insmod命令不再支持-m參數，只有採取其他的方法來獲取模塊加載到內核的地址。通過分析ELF文件格式，我們知道程序中各段的意義如下：

.text（代碼段）：用來存放可執行文件的操作指令，也就是說是它是可執行程序在內存種的鏡像。

.data（數據段）：數據段用來存放可執行文件中已初始化全局變量，也就是存放程序靜態分配的變量和全局變量。

.bss（BSS段）：BSS段包含了程序中未初始化全局變量，在內存中 bss段全部置零。

.rodata（只讀段）：該段保存著只讀數據，在進程映像中構造不可寫的段。

通過在模塊初始化函數中放置一下代碼，我們可以很容易地獲得模塊加載到內存中的地址。

```
1      ……
2      int bss_var;
3      static int hello_init(void)
4      {
5      printk(KERN_ALERT "Text location .text(Code Segment):%p\n",hello_init);
```

```
6       static int data_var=0;
7       printk(KERN_ALERT "Data Location .data(Data Segment):%p\n",&data_var);
8       printk(KERN_ALERT "BSS Location: .bss(BSS Segment):%p\n",&bss_var);
9       ……
10      }
11      Module_init(hello_init);
```

這裡，通過在模塊的初始化函數中添加一段簡單的程序，使模塊在加載時打印出在內核中的加載地址。.rodata段的地址可以通過執行命令readelf -e hello.ko，取得.rodata在文件中的偏移量並加上段的align值得出。

為了使讀者能夠更好地進行模塊的調試，kgdb項目還發佈了一些腳本程序能夠自動探測模塊的插入並自動更新gdb中模塊的符號信息。這些腳本程序的工作原理與前面解釋的工作過程相似，更多的信息請閱讀參考資料[4]。

2.5 硬件斷點

kgdb提供對硬件調試寄存器的支持。在kgdb中可以設置三種硬件斷點：執行斷點（Execution Breakpoint）、寫斷點（Write Breakpoint）、訪問斷點（Access Breakpoint）但不支持I/O訪問的斷點。 目前，kgdb對硬件斷點的支持是通過宏來實現的，最多可以設置4個硬件斷點，這些宏的用法如下：

硬件宏	含义	用法
hwebrk	设置执行断点	hwebrk breakpointno address
hwwbrk	设置写断点	hwwbrk breakpointno length address
hwabrk	设置访问断点	hwabrk breakpointno length address
hwrmbrk	删除断点	hwrmbrk breakpointno
exinfo	显示断点信息	
备注：命令中参数的含义		
Breakpointno：0~3　　length:0~3　　address:十六进制表示的内存地址（省略 0x 前缀）		

表 3：kgdb 硬件断点宏

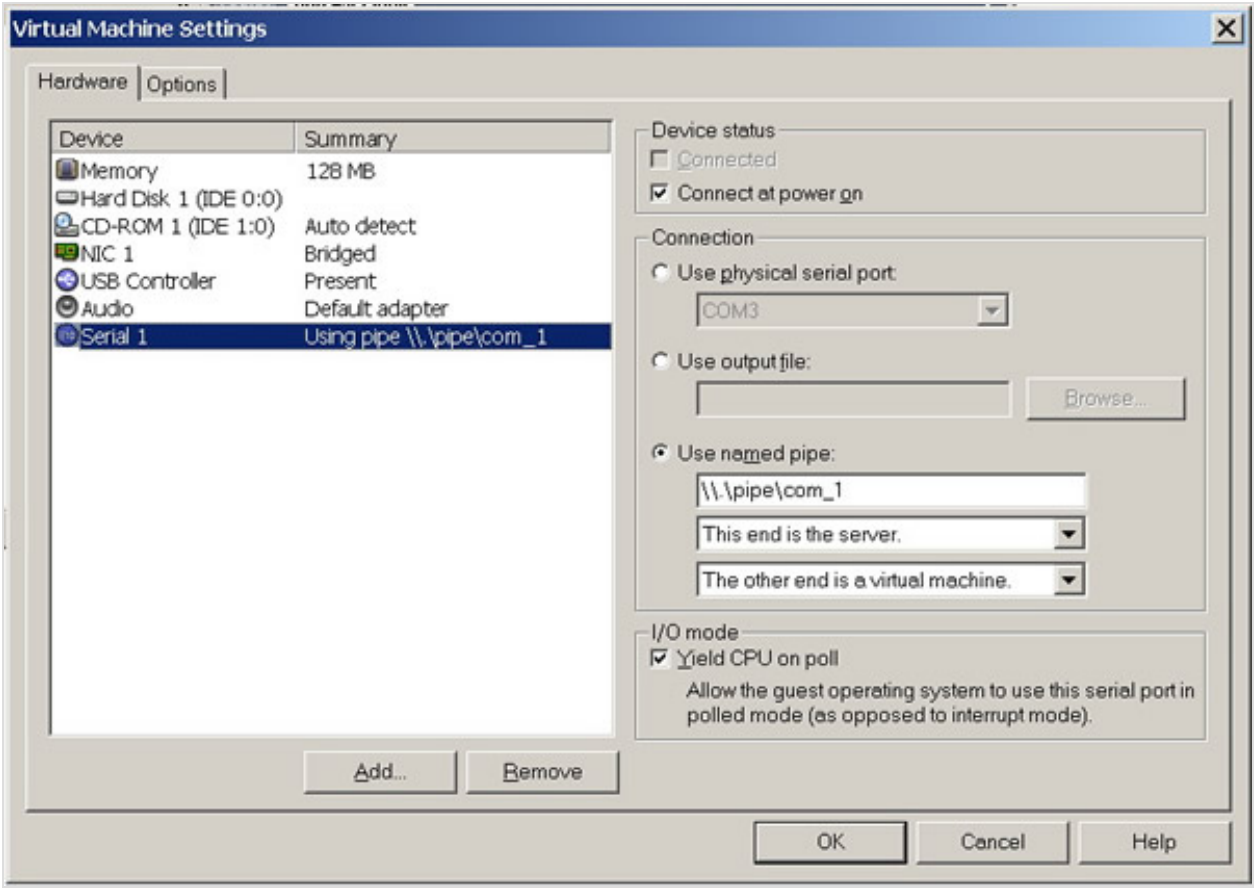
在有些情況下，硬件斷點的使用對於內核的調試是非常方便的。

3 在VMware中搭建調試環境

kgdb調試環境需要使用兩台微機分別充當development機和target機，使用VMware後我們只使用一台計算機就可以順利完成kgdb調試環境的搭建。以windows下的環境為例，創建兩台虛擬機，一台作為開發機，一台作為目標機。

3.1 虛擬機之間的串口連接

虛擬機中的串口連接可以採用兩種方法。一種是指定虛擬機的串口連接到實際的COM上，例如開發機連接到COM1，目標機連接到COM2，然後把兩個串口通過串口線相連接。另一種更為簡便的方法是：在較高一些版本的VMware中都支持把串口映射到命名管道，把兩個虛擬機的串口映射到同一個命名管道。例如，在兩個虛擬機中都選定同一個命名管道 \\.\pipe\com_1,指定target機的COM口為server端，並選擇"The other end is a virtual machine"屬性；指定development機的COM口端為client端，同樣指定COM口的"The other end is a virtual machine"屬性。對於IO mode屬性，在target上選中"Yield CPU on poll"複選擇框，development機不選。這樣，可以無需附加任何硬件，利用虛擬機就可以搭建kgdb調試環境。 即降低了使用kgdb進行調試的硬件要求，也簡化了建立調試環境的過程。



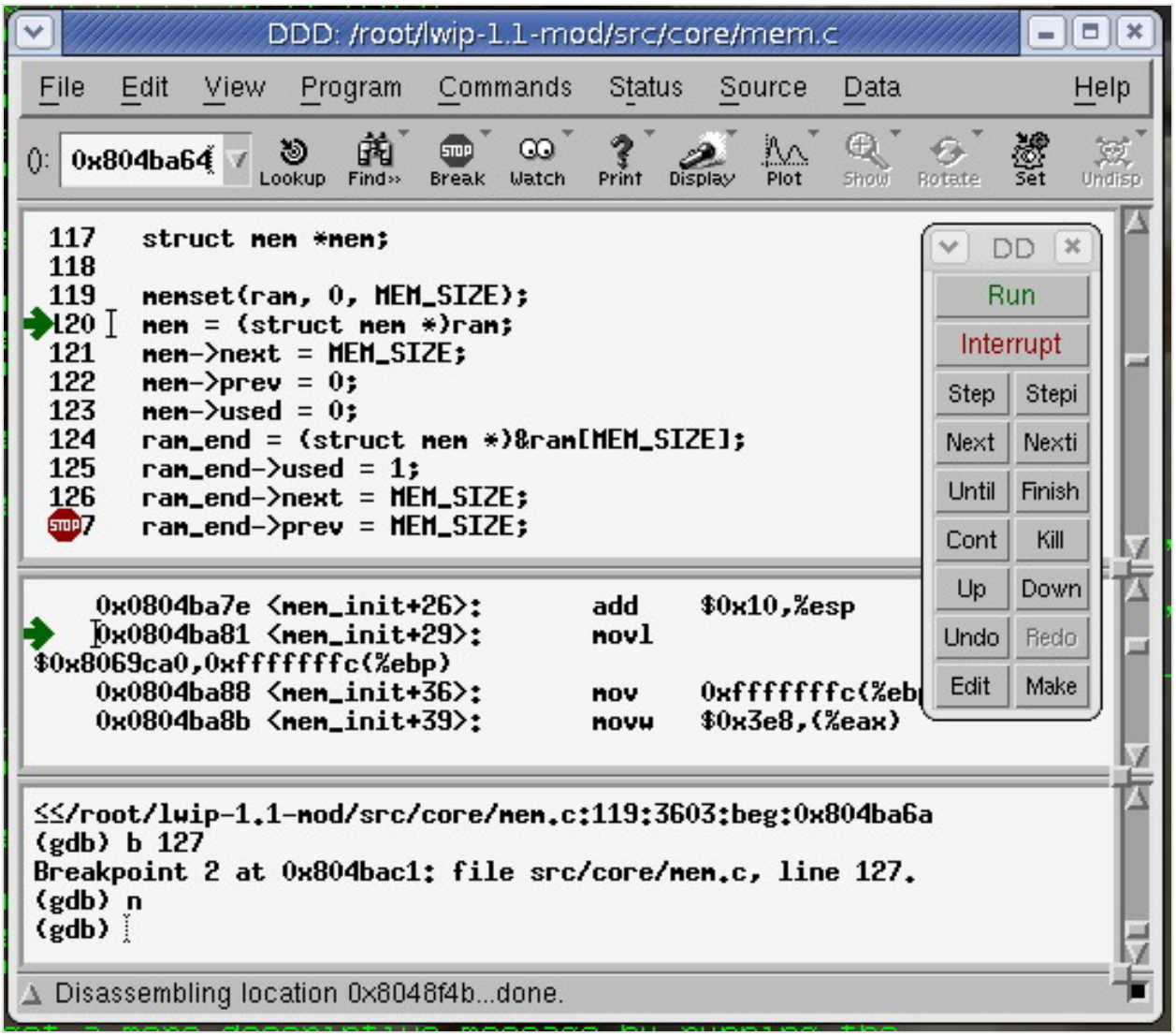
3.2 VMware的使用技巧

VMware虛擬機是比較佔用資源的，尤其是象上面那樣在Windows中使用兩台虛擬機。因此，最好為系統配備512M以上的內存，每台虛擬機至少分配128M的內存。這樣的硬件要求，對目前主流配置的PC而言並不是過高的要求。出於系統性能的考慮，在VMware中儘量使用字符界面進行調試工作。同時，Linux系統默認情況下開啟了sshd服務，建議使用SecureCRT登陸到Linux進行操作，這樣可以有較好的用戶使用界面。

3.3 在Linux下的虛擬機中使用kgdb

對於在Linux下面使用VMware虛擬機的情況，筆者沒有做過實際的探索。從原理上而言，只需要在Linux下只要創建一台虛擬機作為target機，開發機的工作可以在實際的Linux環境中進行，搭建調試環境的過程與上面所述的過程類似。由於只需要創建一台虛擬機，所以使用Linux下的虛擬機搭建kgdb調試環境對系統性能的要求較低。（vmware

已經推出了Linux下的版本）還可以在development機上配合使用一些其他的調試工具，例如功能更強大的cgdb、圖形界面的DDD調試器等，以方便內核的調試工作。



4 kgdb的一些特點和不足

使用kgdb作為內核調試環境最大的不足在於對kgdb硬件環境的要求較高，必須使用兩台計算機分別作為target和development機。儘管使用虛擬機的方法可以只用一台PC即能搭建調試環境，但是對系統其他方面的性能也提出了一定的要求，同時也增加了搭建調試環境時複雜程度。另外，kgdb內核的編譯、配置也比較複雜，需要一定的技巧，筆

者當時做的時候也是費了很多周折。當調試過程結束後時，還需要重新製作所要發佈的內核。使用kgdb並不能進行全程調試，也就是說kgdb並不能用於調試系統一開始的初始化引導過程。

不過，kgdb是一個不錯的內核調試工具，使用它可以進行對內核的全面調試，甚至可以調試內核的中斷處理程序。如果在一些圖形化的開發工具的幫助下，對內核的調試將更方便。

參考：

透過虛擬化技術體驗kgdb

Linux 系統內核的調試

Debugging The Linux Kernel Using Gdb

十 使用SkyEye構建Linux內核調試環境

SkyEye是一個開源軟件項目（OPenSource Software）,SkyEye項目的目標是在通用的Linux和Windows平台上模擬常見的嵌入式計算機系統。SkyEye實現了一個指令級的硬件模擬平台，可以模擬多種嵌入式開發板，支持多種CPU指令集。SkyEye 的核心是 GNU 的 gdb 項目，它把gdb和 ARM Simulator很好地結合在了一起。加入ARMulator的功能之後，它就可以來仿真嵌入式開發板，在它上面不僅可以調試硬件驅動，還可以調試操作系統。Skyeye項目目前已經在嵌入式系統開發領域得到了很大的推廣。

1 SkyEye的安裝和μcLinux內核編譯

1.1 SkyEye的安裝

SkyEye的安裝不是本文要介紹的重點，目前已經有大量的資料對此進行了介紹。有關SkyEye的安裝與使用的內容請查閱參考資料[11]。由於skyeye面目主要用於嵌入式系統領域，所以在skyeye上經常使用的是μcLinux系統，當然使用Linux作為skyeye上運行的系統也是可以的。由於介紹μcLinux 2.6在skyeye上編譯的相關資料並不多，所以下面進行詳細介紹。

1.2 μcLinux 2.6.x的編譯

要在SkyEye中調試操作系統內核，首先必須使被調試內核能在SkyEye所模擬的開發板上正確運行。因此，正確編譯待調試操作系統內核並配置SkyEye是進行內核調試的第一步。下面我們以SkyEye模擬基於Atmel AT91X40的開發板，並運行μcLinux 2.6為例介紹SkyEye的具體調試方法。

I、安裝交叉編譯環境

先安裝交叉編譯器。儘管在一些資料中説明使用工具鏈arm-elf-tools-20040427.sh ,但是由於arm-elf-xxx與arm-linux-xxx對宏及鏈接處理的不同，經驗證明使用arm-elf-xxx工具鏈在鏈接vmlinux的最後階段將會出錯。所以這裡我們使用的交叉編譯工具鏈是：arm-uclinux-tools-base-gcc3.4.0-20040713.sh，關於該交叉編譯工具鏈的下載地址請參見[6]。注意以下步驟最好用root用戶來執行。

```
1    [root@lisl tmp]#chmod +x  arm-uclinux-tools-base-gcc3.4.0-20040713.sh
2    [root@lisl tmp]#./arm-uclinux-tools-base-gcc3.4.0-20040713.sh
```

安裝交叉編譯工具鏈之後，請確保工具鏈安裝路徑存在於系統PATH變量中。

II、製作μcLinux內核

得到μcLinux發佈包的一個最容易的方法是直接訪問uClinux.org站點[7]。該站點發佈的內核版本可能不是最新的，但你能找到一個最新的μcLinux補丁以及找一個對應的Linux內核版本來製作一個最新的μcLinux內核。這裡，將使用這種方法來製作最新的μcLinux內核。目前（筆者記錄編寫此文章時），所能得到的發布包的最新版本是uClinux-dist.20041215.tar.gz。

下載uClinux-dist.20041215.tar.gz，文件的下載地址請參見[7]。

下載linux-2.6.9-hsc0.patch.gz，文件的下載地址請參見[8]。

下載linux-2.6.9.tar.bz2，文件的下載地址請參見[9]。

現在我們得到了整個的linux-2.6.9源代碼，以及所需的內核補丁。請準備一個有2GB空間的目錄裡來完成以下製作μcLinux內核的過程。

```
1    [root@lisl tmp]# tar -jxvf uClinux-dist-20041215.tar.bz2
2    [root@lisl uClinux-dist]# tar -jxvf  linux-2.6.9.tar.bz2
3    [root@lisl uClinux-dist]# gzip -dc linux-2.6.9-hsc0.patch.gz | patch -p0
```

或者使用：

```
1    [root@lisl uClinux-dist]# gunzip linux-2.6.9-hsc0.patch.gz
2    [root@lisl uClinux-dist]patch -p0 < linux-2.6.9-hsc0.patch
```

執行以上過程後，將在linux-2.6.9/arch目錄下生成一個補丁目錄－armnommu。刪除原來μcLinux目錄裡的linux-2.6.x(即那個linux-2.6.9-uc0)，並將我們打好補丁的Linux內核目錄更名為linux-2.6.x。

```
1    [root@lisl uClinux-dist]# rm -rf linux-2.6.x/
2    [root@lisl uClinux-dist]# mv linux-2.6.9 linux-2.6.x
```

III、配置和編譯μcLinux內核

因為只是出於調試μcLinux內核的目的，這裡沒有生成uClibc庫文件及romfs.img文件。在發佈μcLinux時，已經預置了某些常用嵌入式開發板的配置文件，因此這裡直接使用這些配置文件，過程如下：

```
1    [root@lisl uClinux-dist]# cd linux-2.6.x
2    [root@lisl linux-2.6.x]#make ARCH=armnommu CROSS_COMPILE=arm-uclinux- atmel_deconfig
```

atmel_deconfig文件是μcLinux發佈時提供的一個配置文件，存放於目錄linux-2.6.x /arch/armnommu/configs/中。

```
1    [root@lisl linux-2.6.x]#make ARCH=armnommu CROSS_COMPILE=arm-uclinux- oldconfig
```

下面編譯配置好的內核：

```
1    [root@lisl linux-2.6.x]# make ARCH=armnommu CROSS_COMPILE=arm-uclinux- v=1
```

一般情況下，編譯將順利結束並在Linux-2.6.x/目錄下生成未經壓縮的μcLinux內核文件vmlinux。需要注意的是為了調試μcLinux內核，需要打開內核編譯的調試選項-g，使編譯後的內核帶有調試信息。打開編譯選項的方法可以選擇：

"Kernel debugging->Compile the kernel with debug info"後將自動打開調試選項。也可以直接修改linux-2.6.x目錄下的Makefile文件，為其打開調試開關。方法如下：。

```
1    CFLAGS  += -g
```

最容易出現的問題是找不到arm-uclinux-gcc命令的錯誤，主要原因是PATH變量中沒有 包含arm-uclinux-gcc命令所在目錄。在arm-linux-gcc的缺省安裝情況下，它的安裝目錄是/root/bin/arm-linux-tool/，使用以下命令將路徑加到PATH環境變量中。

```
1     Export PATH=$PATH:/root/bin/arm-linux-tool/bin
```

IV、根文件系統的製作

Linux內核在啟動的時的最後操作之一是加載根文件系統。根文件系統中存放了嵌入式 系統使用的所有應用程序、庫文件及其他一些需要用到的服務。出於文章篇幅的考慮，這裡不打算介紹根文件系統的製作方法，讀者可以查閱一些其他的相關資料。值得注意的是，由配置文件skyeye.conf指定了裝載到內核中的根文件系統。

2 使用SkyEye調試

編譯完μcLinux內核後，就可以在SkyEye中調試該ELF執行文件格式的內核了。前面已經說過利用SkyEye調試內核與使用gdb調試運用程序的方法相同。

需要提醒讀者的是，SkyEye的配置文件－skyeye.conf記錄了模擬的硬件配置和模擬執行行為。該配置文件是SkyEye系統中一個及其重要的文件，很多錯誤和異常情況的發生都和該文件有關。在安裝配置SkyEye出錯時，請首先檢查該配置文件然後再進行其他的工作。此時，所有的準備工作已經完成，就可以進行內核的調試工作了。

3 使用SkyEye調試內核的特點和不足

在SkyEye中可以進行對Linux系統內核的全程調試。由於SkyEye目前主要支持基於ARM內核的CPU，因此一般而言需要使用交叉編譯工具編譯待調試的Linux系統內核。另外，製作SkyEye中使用的內核編譯、配置過程比較複雜、繁瑣。不過，當調試過程結束後無需重新製作所要發佈的內核。

SkyEye只是對系統硬件進行了一定程度上的模擬，所以在SkyEye與真實硬件環境相比較而言還是有一定的差距，這對一些與硬件緊密相關的調試可能會有一定的影響，例如驅動程序的調試。不過對於大部分軟件的調試，SkyEye已經提供了精度足夠的模擬了。

SkyEye的下一個目標是和eclipse結合，有了圖形界面，能為調試和查看源碼提供一些方便。

參考：

Linux 系統內核的調試

十一 KDB

Linux 內核調試器（KDB）允許您調試 Linux 內核。這個恰如其名的工具實質上是內核代碼的補丁，它允許高手訪問內核內存和數據結構。KDB 的主要優點之一就是它不需要用另一台機器進行調試：您可以調試正在運行的內核。

設置一台用於 KDB 的機器需要花費一些工作，因為需要給內核打補丁並進行重新編譯。KDB 的用戶應當熟悉 Linux 內核的編譯（在一定程度上還要熟悉內核內部機理）。

在本文中，我們將從有關下載 KDB 補丁、打補丁、（重新）編譯內核以及啟動 KDB 方面的信息著手。然後我們將瞭解 KDB 命令並研究一些較常用的命令。最後，我們將研究一下有關設置和顯示選項方面的一些詳細信息。

1 入門

KDB 項目是由 Silicon Graphics 維護的，您需要從它的 FTP 站點下載與內核版本有關的補丁。（在編寫本文時）可用的最新 KDB 版本是 4.2。您將需要下載並應用兩個補丁。一個是「公共的」補丁，包含了對通用內核代碼的更改，另一個是特定於體系結構的補丁。補丁可作為 bz2 文件獲取。例如，在運行 2.4.20 內核的 x86 機器上，您會需要 kdb-v4.2-2.4.20-common-1.bz2 和 kdb-v4.2-2.4.20-i386-1.bz2。

這裡所提供的所有示例都是針對 i386 體系結構和 2.4.20 內核的。您將需要根據您的機器和內核版本進行適當的更改。您還需要擁有 root 許可權以執行這些操作。

將文件複製到 /usr/src/linux 目錄中並從用 bzip2 壓縮的文件解壓縮補丁文件：

```
1     #bzip2 -d kdb-v4.2-2.4.20-common-1.bz2
2     #bzip2 -d kdb-v4.2-2.4.20-i386-1.bz2
```

您將獲得 kdb-v4.2-2.4.20-common-1 和 kdb-v4.2-2.4-i386-1 文件。

現在，應用這些補丁：

```
1     #patch -p1 <kdb-v4.2-2.4.20-common-1
2     #patch -p1 <kdb-v4.2-2.4.20-i386-1
```

這些補丁應該乾淨利落地加以應用。查找任何以 .rej 結尾的文件。這個擴展名表明這些是失敗的補丁。如果內核樹沒問題，那麼補丁的應用就不會有任何問題。

接下來，需要構建內核以支持 KDB。第一步是設置 CONFIG_KDB 選項。使用您喜歡的配置機制（xconfig 和 menuconfig 等）來完成這一步。轉到結尾處的「Kernel hacking」部分並選擇「Built-in Kernel Debugger support」選項。

您還可以根據自己的偏好選擇其它兩個選項。選擇「Compile the kernel with frame pointers」選項（如果有的話）則設置CONFIG_FRAME_POINTER 標誌。這將產生更好的堆棧回溯，因為幀指針寄存器被用作幀指針而不是通用寄存器。您還可以選擇「KDB off by default」選項。這將設置 CONFIG_KDB_OFF 標誌，並且在缺省情況下將關閉 KDB。我們將在後面一節中對此進行詳細介紹。

保存配置，然後退出。重新編譯內核。建議在構建內核之前執行「make clean」。用常用方式安裝內核並引導它。

2 初始化並設置環境變量

您可以定義將在 KDB 初始化期間執行的 KDB 命令。需要在純文本文件 kdb_cmds 中定義這些命令，該文件位於 Linux 源代碼樹（當然是在打了補丁之後）的 KDB 目錄中。該文件還可以用來定義設置顯示和打印選項的環境變量。文件開頭的註釋提供了編輯文件方面的幫助。使用這個文件的缺點是，在您更改了文件之後需要重新構建並重新安裝內核。

3 激活 KDB

如果編譯期間沒有選中 CONFIG_KDB_OFF ，那麼在缺省情況下 KDB 是活動的。否則，您需要顯式地激活它 — 通過在引導期間將kdb=on 標誌傳遞給內核或者通過在掛裝了 /proc 之後執行該工作：

```
1     #echo "1" >/proc/sys/kernel/kdb
```

倒過來執行上述步驟則會取消激活 KDB。也就是說，如果缺省情況下 KDB 是打開的，那麼將 kdb=off 標誌傳遞給內核或者執行下面這個操作將會取消激活 KDB：

```
1     #echo "0" >/proc/sys/kernel/kdb
```

在引導期間還可以將另一個標誌傳遞給內核。 kdb=early 標誌將導致在引導過程的初始階段就把控制權傳遞給 KDB。如果您需要在引導過程初始階段進行調試，那麼這將有所幫助。

調用 KDB 的方式有很多。如果 KDB 處於打開狀態，那麼只要內核中有緊急情況就自動調用它。按下鍵盤上的 PAUSE 鍵將手工調用 KDB。調用 KDB 的另一種方式是通過串行控制台。當然，要做到這一點，需要設置串行控制台並且需要一個從串行控制台進行讀取的程序。按鍵序列 Ctrl-A 將從串行控制台調用 KDB。

4 KDB 命令

KDB 是一個功能非常強大的工具，它允許進行幾個操作，比如內存和寄存器修改、應用斷點和堆棧跟蹤。根據這些，可以將 KDB 命令分成幾個類別。下面是有關每一類中最常用命令的詳細信息。

4.1 內存顯示和修改

這一類別中最常用的命令是 md 、 mdr 、 mm 和 mmW 。

md 命令以一個地址／符號和行計數為參數，顯示從該地址開始的 line-count 行的內存。如果沒有指定 line-count ，那麼就使用環境變量所指定的缺省值。如果沒有指定地址，那麼 md 就從上一次打印的地址繼續。地址打印在開頭，字符轉換打印在結尾。

mdr 命令帶有地址／符號以及字節計數，顯示從指定的地址開始的 byte-count 字節數的初始內存內容。它本質上和 md 一樣，但是它不顯示起始地址並且不在結尾顯示字符轉換。 mdr 命令較少使用。

mm 命令修改內存內容。它以地址／符號和新內容作為參數，用 new-contents 替換地址處的內容。

mmW 命令更改從地址開始的 W 個字節。請注意， mm 更改一個機器字。

示例

顯示從 0xc000000 開始的 15 行內存：

```
1     [0]kdb> md 0xc000000 15
```

將內存位置為 0xc000000 上的內容更改為 0x10：

```
1     [0]kdb> mm 0xc000000 0x10
```

4.2 寄存器顯示和修改

這一類別中的命令有 rd 、 rm 和 ef 。

rd 命令（不帶任何參數）顯示處理器寄存器的內容。它可以有選擇地帶三個參數。如果傳遞了 c 參數，則 rd 顯示處理器的控制寄存器；如果帶有 d 參數，那麼它就顯示調試寄存器；如果帶有 u 參數，則顯示上一次進入內核的當前任務的寄存器組。

rm 命令修改寄存器的內容。它以寄存器名稱和 new-contents 作為參數，用 new-contents 修改寄存器。寄存器名稱與特定的體系結構有關。目前，不能修改控制寄存器。

ef 命令以一個地址作為參數，它顯示指定地址處的異常幀。

示例

顯示通用寄存器組：

```
1    [0]kdb> rd
2    [0]kdb> rm %ebx 0x25
```

4.3 斷點

常用的斷點命令有 bp 、 bc 、 bd 、 be 和 bl 。

bp 命令以一個地址／符號作為參數，它在地址處應用斷點。當遇到該斷點時則停止執行並將控制權交予 KDB。該命令有幾個有用的變體。 bpa 命令對 SMP 系統中的所有處理器應用斷點。 bph 命令強制在支持硬件寄存器的系統上使用它。 bpha 命令類似於 bpa 命令，差別在於它強制使用硬件寄存器。

bd 命令禁用特殊斷點。它接收斷點號作為參數。該命令不是從斷點表中除去斷點，而只是禁用它。斷點號從 0 開始，根據可用性順序分配給斷點。

be 命令啟用斷點。該命令的參數也是斷點號。

bl 命令列出當前的斷點集。它包含了啟用的和禁用的斷點。

bc 命令從斷點表中除去斷點。它以具體的斷點號或 * 作為參數，在後一種情況下它將除去所有斷點。

示例

對函數 sys_write() 設置斷點：

```
1    [0]kdb> bp sys_write
```

列出斷點表中的所有斷點：

```
1    [0]kdb> bl
```

清除斷點號 1：

```
1    [0]kdb> bc 1
```

4.4 堆棧跟蹤

主要的堆棧跟蹤命令有 bt 、 btp 、 btc 和 bta 。

bt 命令設法提供有關當前線程的堆棧的信息。它可以有選擇地將堆棧幀地址作為參數。如果沒有提供地址，那麼它採用當前寄存器來回溯堆棧。否則，它假定所提供的地址是有效的堆棧幀起始地址並設法進行回溯。如果內核編譯期間設置了CONFIG_FRAME_POINTER 選項，那麼就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。如果沒有設置CONFIG_FRAME_POINTER ，那麼 bt 命令可能會產生錯誤的結果。

btp 命令將進程標識作為參數，並對這個特定進程進行堆棧回溯。

btc 命令對每個活動 CPU 上正在運行的進程執行堆棧回溯。它從第一個活動 CPU 開始執行 bt ，然後切換到下一個活動 CPU，以此類推。

bta 命令對處於某種特定狀態的所有進程執行回溯。若不帶任何參數，它就對所有進程執行回溯。可以有選擇地將各種參數傳遞給該命令。將根據參數處理處於特定狀態的進程。選項以及相應的狀態如下：

- D：不可中斷狀態
- R：正運行
- S：可中斷休眠
- T：已跟蹤或已停止
- Z：僵死
- U：不可運行

這類命令中的每一個都會打印出一大堆信息。示例

跟蹤當前活動線程的堆棧：

```
1    [0]kdb> bt
```

跟蹤標識為 575 的進程的堆棧：

```
1    [0]kdb> btp 575
```

4.5 其它命令

下面是在內核調試過程中非常有用的其它幾個 KDB 命令。

id 命令以一個地址／符號作為參數，它對從該地址開始的指令進行反彙編。環境變量 IDCOUNT 確定要顯示多少行輸出。

ss 命令單步執行指令然後將控制返回給 KDB。該指令的一個變體是 ssb ，它執行從當前指令指針地址開始的指令（在屏幕上打印指令），直到它遇到將引起分支轉移的指令為止。分支轉移指令的典型示例有 call 、 return 和 jump 。

go 命令讓系統繼續正常執行。一直執行到遇到斷點為止（如果已應用了一個斷點的話）。

reboot 命令立刻重新引導系統。它並沒有徹底關閉系統，因此結果是不可預測的。

ll 命令以地址、偏移量和另一個 KDB 命令作為參數。它對鏈表中的每個元素反覆執行作為參數的這個命令。所執行的命令以列表中當前元素的地址作為參數。

示例

反彙編從例程 schedule 開始的指令。所顯示的行數取決於環境變量 IDCOUNT ：

```
1     [0]kdb> id schedule
```

執行指令直到它遇到分支轉移條件（在本例中為指令 jne ）為止：

```
1     [0]kdb> ssb
2     0xc0105355  default_idle+0x25:  cli
3     0xc0105356  default_idle+0x26:  mov  0x14(%edx),%eax
4     0xc0105359  default_idle+0x29:  test %eax, %eax
5     0xc010535b  default_idle+0x2b:  jne  0xc0105361 default_idle+0x31
```

5 技巧和訣竅

調試一個問題涉及到：使用調試器（或任何其它工具）找到問題的根源以及使用源代碼來跟蹤導致問題的根源。單單使用源代碼來確定問題是極其困難的，只有老練的內核黑客才有可能做得到。相反，大多數的新手往往要過多地依靠調試器來修正錯誤。這種方法可能會產生不正確的問題解決方案。我們擔心的是這種方法只會修正表面症狀而不能解決真正的問題。此類錯誤的典型示例是添加錯誤處理代碼以處理 NULL 指針或錯誤的引用，卻沒有查出無效引用的真正原因。

結合研究代碼和使用調試工具這兩種方法是識別和修正問題的最佳方案。

調試器的主要用途是找到錯誤的位置、確認症狀（在某些情況下還有起因）、確定變量的值，以及確定程序是如何出現這種情況的（即，建立調用堆棧）。有經驗的黑客會知道對於某種特定的問題應使用哪一個調試器，並且能迅速地根據調試獲取必要的信息，然後繼續分析代碼以識別起因。

因此，這裡為您介紹了一些技巧，以便您能使用 KDB 快速地取得上述結果。當然，要記住，調試的速度和精確度來自經驗、實踐和良好的系統知識（硬件和內核內部機理等）。

5.1 技巧 #1

在 KDB 中，在提示處輸入地址將返回與之最為匹配的符號。這在堆棧分析以及確定全局數據的地址／值和函數地址方面極其有用。同樣，輸入符號名則返回其虛擬地址。

示例

表明函數 sys_read 從地址 0xc013db4c 開始：

```
1     [0]kdb> 0xc013db4c
2     0xc013db4c = 0xc013db4c (sys_read)
```

同樣，表明 sys_write 位於地址 0xc013dcc8：

```
1     [0]kdb> sys_write
2     sys_write = 0xc013dcc8 (sys_write)
```

這些有助於在分析堆棧時找到全局數據和函數地址。

5.2 技巧 #2

在編譯帶 KDB 的內核時，只要 CONFIG_FRAME_POINTER 選項出現就使用該選項。為此，需要在配置內核時選擇「Kernel hacking」部分下面的「Compile the kernel with frame pointers」選項。這確保了幀指針寄存器將被用作幀指針，從而產生正確的回溯。實際上，您可以手工轉儲幀指針寄存器的內容並跟蹤整個堆棧。例如，在 i386 機器上，%ebp 寄存器可以用來回溯整個堆棧。

例如，在函數 rmqueue() 上執行第一個指令後，堆棧看上去類似於下面這樣：

```
1      [0]kdb> md %ebp
```

```
2      0xc74c9f38 c74c9f60  c0136c40 000001f0 00000000
3      0xc74c9f48 08053328 c0425238 c04253a8 00000000
4      0xc74c9f58 000001f0  00000246 c74c9f6c c0136a25
5      0xc74c9f68 c74c8000  c74c9f74  c0136d6d c74c9fbc
6      0xc74c9f78 c014fe45  c74c8000  00000000 08053328
7      [0]kdb> 0xc0136c40
8      0xc0136c40 = 0xc0136c40 (__alloc_pages +0x44)
9      [0]kdb> 0xc0136a25
10     0xc0136a25 = 0xc0136a25 (_alloc_pages +0x19)
11     [0]kdb> 0xc0136d6d
12     0xc0136d6d = 0xc0136d6d (__get_free_pages +0xd)
```

我們可以看到 rmqueue() 被 ___alloc_pages 調用，後者接下來又被 _alloc_pages 調用，以此類推。

每一幀的第一個雙字（double word）指向下一幀，這後面緊跟著調用函數的地址。因此，跟蹤堆棧就變成一件輕鬆的工作了。

5.3 技巧 #3

go 命令可以有選擇地以一個地址作為參數。如果您想在某個特定地址處繼續執行，則可以提供該地址作為參數。另一個辦法是使用rm 命令修改指令指針寄存器，然後只要輸入 go 。如果您想跳過似乎會引起問題的某個特定指令或一組指令，這就會很有用。但是，請注意，該指令使用不慎會造成嚴重的問題，系統可能會嚴重崩潰。

5.4 技巧 #4

您可以利用一個名為 defcmd 的有用命令來定義自己的命令集。例如，每當遇到斷點時，您可能希望能同時檢查某個特殊變量、檢查某些寄存器的內容並轉儲堆棧。通常，您必須要輸入一系列命令，以便能同時執行所有這些工作。defcmd 允許您定義自己的命令，該命令可以包含一個或多個預定義的 KDB 命令。然後只需要用一個命令就可以完成所有這三項工作。其語法如下：

```
1     [0]kdb> defcmd name "usage" "help"
2     [0]kdb> [defcmd] type the commands here
3     [0]kdb> [defcmd] endefcmd
```

例如，可以定義一個（簡單的）新命令 hari ，它顯示從地址 0xc000000 開始的一行內存、顯示寄存器的內容並轉儲堆棧：

```
1     [0]kdb> defcmd hari "" "no arguments needed"
2     [0]kdb> [defcmd] md 0xc000000 1
3     [0]kdb> [defcmd] rd
4     [0]kdb> [defcmd] md %ebp 1
5     [0]kdb> [defcmd] endefcmd
```

該命令的輸出會是：

```
1      [0]kdb> hari
2      [hari]kdb> md 0xc000000 1
3      0xc000000 00000001 f000e816 f000e2c3 f000e816
4      [hari]kdb> rd
5      eax = 0x00000000 ebx = 0xc0105330 ecx = 0xc0466000 edx = 0xc0466000
6      ....
7      ...
8      [hari]kdb> md %ebp 1
9      0xc0467fbc c0467fd0 c01053d2 00000002 000a0200
10     [0]kdb>
```

5.5 技巧 #5

可以使用 bph 和 bpha 命令（假如體系結構支持使用硬件寄存器）來應用讀寫斷點。這意味著每當從某個特定地址讀取數據或將數據寫入該地址時，我們都可以對此進行控制。當調試數據／內存毀壞問題時這可能會極其方便，在這種情況中您可以用它來識別毀壞的代碼／進程。

示例

每當將四個字節寫入地址 0xc0204060 時就進入內核調試器：

```
1     [0]kdb> bph 0xc0204060 dataw 4
```

在讀取從 0xc000000 開始的至少兩個字節的數據時進入內核調試器：

```
1     [0]kdb> bph 0xc000000 datar 2
```

6 結束語

對於執行內核調試，KDB 是一個方便的且功能強大的工具。它提供了各種選項，並且使我們能夠分析內存內容和數據結構。最妙的是，它不需要用另一台機器來執行調試。

參考：

Linux 內核調試器內幕 KDB入門指南

十二 Kprobes

Kprobes 是 Linux 中的一個簡單的輕量級裝置，讓您可以將斷點插入到正在運行的內核之中。 Kprobes 提供了一個強行進入任何內核例程並從中斷處理器無干擾地收集信息的接口。使用 Kprobes 可以 輕鬆地收集處理器寄存器和全局數據結構等調試信息。開發者甚至可以使用 Kprobes 來修改 寄存器值和全局數據結構的值。

為完成這一任務，Kprobes 向運行的內核中給定地址寫入斷點指令，插入一個探測器。 執行被探測的指令會導致斷點錯誤。Kprobes 鉤住（hook in）斷點處理器並收集調試信息。Kprobes 甚至可以單步執行被探測的指令。

1 安裝

要安裝 Kprobes，需要從 Kprobes 主頁下載最新的補丁。 打包的文件名稱類似於 kprobes-2.6.8-rc1.tar.gz。解開補丁並將其安裝到 Linux 內核：

```
1    $tar -xvzf kprobes-2.6.8-rc1.tar.gz
2    $cd /usr/src/linux-2.6.8-rc1
3    $patch -p1 < ../kprobes-2.6.8-rc1-base.patch
```

Kprobes 利用了 SysRq 鍵，這個 DOS 時代的產物在 Linux 中有了新的用武之地。您可以在 Scroll Lock鍵左邊找到 SysRq 鍵；它通常標識為 Print Screen。要為 Kprobes 啟用 SysRq 鍵，需要安裝 kprobes-2.6.8-rc1-sysrq.patch 補丁：

```
1    $patch -p1 < ../kprobes-2.6.8-rc1-sysrq.patch
```

使用 make xconfig/ make menuconfig/ make oldconfig 配置內核，並 啟用 CONFIG_KPROBES 和 CONFIG_MAGIC_SYSRQ標記。 編譯並引導到新內核。您現在就已經準備就緒，可以插入 printk 並通過編寫簡單的 Kprobes 模塊來動態而且無干擾地 收集調試信息。

2 編寫 Kprobes 模塊

對於每一個探測器，您都要分配一個結構體 struct kprobe kp; （參考 include/linux/kprobes.h 以獲得關於此數據結構的詳細信息）。

清單 9. 定義 pre、post 和 fault 處理器

```
1     /* pre_handler: this is called just before the probed instruction is
2      * executed.
3      */
4     int handler_pre(struct kprobe *p, struct pt_regs *regs) {
5     printk("pre_handler: p->addr=0x%p, eflags=0x%lx\n",p->addr,
6     regs->eflags);
7     return 0;
8     }
9     /* post_handler: this is called after the probed instruction is executed
10     * (provided no exception is generated).
11     */
12    void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags) {
13    printk("post_handler: p->addr=0x%p, eflags=0x%lx \n", p->addr,
14    regs->eflags);
15    }
16    /* fault_handler: this is called if an exception is generated for any
17     * instruction within the fault-handler, or when Kprobes
18     * single-steps the probed instruction.
19     */
20    int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr) {
```

```
21     printk("fault_handler:p->addr=0x%p, eflags=0x%lx\n", p->addr,
22     regs->eflags);
23     return 0;
24     }
```

2.1 獲得內核例程的地址

在註冊過程中，您還需要指定插入探測器的內核例程的地址。使用這些方法中的任意一個來獲得內核例程 的地址：

1. 從 System.map 文件直接得到地址。
2. 例如，要得到 do_fork 的地址，可以在命令行執行 $grep do_fork /usr/src/linux/System.map 。
3. 使用 nm 命令。
4. $nm vmlinuz |grep do_fork
5. 從 /proc/kallsyms 文件獲得地址。
6. $cat /proc/kallsyms |grep do_fork
7. 使用 kallsyms_lookup_name() 例程。
8. 這個例程是在 kernel/kallsyms.c 文件中定義的，要使用它，必須啟用 CONFIG_KALLSYMS 編譯內核。kallsyms_lookup_name() 接受一個字符串格式內核例程名， 返回那個內核例程的地址。例如：kallsyms_lookup_name("do_fork");

然後在 init_moudle 中註冊您的探測器：

清單 10. 註冊一個探測器

```
1      /* specify pre_handler address
2       */
3      kp.pre_handler=handler_pre;
4      /* specify post_handler address
5       */
6      kp.post_handler=handler_post;
7      /* specify fault_handler address
8       */
9      kp.fault_handler=handler_fault;
10     /* specify the address/offset where you want to insert probe.
11      * You can get the address using one of the methods described above.
12      */
```

```
13     kp.addr = (kprobe_opcode_t *) kallsyms_lookup_name("do_fork");
14     /* check if the kallsyms_lookup_name() returned the correct value.
15      */
16     if (kp.add == NULL) {
17     printk("kallsyms_lookup_name could not find address
18     for the specified symbol name\n");
19     return 1;
20     }
21     /* or specify address directly.
22      * $grep "do_fork" /usr/src/linux/System.map
23      * or
24      * $cat /proc/kallsyms |grep do_fork
25      * or
26      * $nm vmlinuz |grep do_fork
27      */
28     kp.addr = (kprobe_opcode_t *) 0xc01441d0;
29     /* All set to register with Kprobes
30      */
31            register_kprobe(&kp);
```

一旦註冊了探測器，運行任何 shell 命令都會導致一個對 do_fork 的調用，您將可以在控制台上或者運行 dmesg 命令來查看您的 printk。做完後要記得註銷探測器：

unregister_kprobe(&kp);

下面的輸出顯示了 kprobe 的地址以及 eflags 寄存器的內容：

$tail -5 /var/log/messages

Jun 14 18:21:18 llm05 kernel: pre_handler: p->addr=0xc01441d0, eflags=0x202

Jun 14 18:21:18 llm05 kernel: post_handler: p->addr=0xc01441d0, eflags=0x196

2.2 獲得偏移量

您可以在例程的開頭或者函數中的任意偏移位置插入 printk（偏移量必須在指令範圍之內）。 下面的代碼示例展示了如何來計算偏移量。首先，從對象文件中反彙編機器指令，並將它們 保存為一個文件：

```
1     $objdump -D /usr/src/linux/kernel/fork.o > fork.dis
```

其結果是：

清單 11. 反彙編的 fork

```
1      000022b0 <do_fork>:
2         22b0:       55                      push   %ebp
3         22b1:       89 e5                   mov    %esp,%ebp
4         22b3:       57                      push   %edi
5         22b4:       89 c7                   mov    %eax,%edi
6         22b6:       56                      push   %esi
7         22b7:       89 d6                   mov    %edx,%esi
8         22b9:       53                      push   %ebx
9         22ba:       83 ec 38                sub    $0x38,%esp
10        22bd:       c7 45 d0 00 00 00 00    movl   $0x0,0xffffffd0(%ebp)
11        22c4:       89 cb                   mov    %ecx,%ebx
12        22c6:       89 44 24 04             mov    %eax,0x4(%esp)
13        22ca:       c7 04 24 0a 00 00 00    movl   $0xa,(%esp)
14        22d1:       e8 fc ff ff ff          call   22d2 <do_fork+0x22>
15        22d6:       b8 00 e0 ff ff          mov    $0xffffe000,%eax
16        22db:       21 e0                   and    %esp,%eax
17        22dd:       8b 00                   mov    (%eax),%eax
```

要在偏移位置 0x22c4 插入探測器，先要得到與例程的開始處相對的偏移量 0x22c4 - 0x22b0 = 0x14 ，然後將這個偏移量添加到 do_fork 的地址 0xc01441d0 + 0x14 。（運行 $cat /proc/kallsyms | grep do_fork 命令以獲得 do_fork 的地址。）

您還可以將 do_fork 的相對偏移量 0x22c4 - 0x22b0 = 0x14 添加到 kallsyms_lookup_name("do_fork"); 的輸入，即：0x14 + kallsyms_lookup_name("do_fork");

2.3 轉儲內核數據結構

現在，讓我們使用修改過的用來轉儲數據結構的 Kprobe post_handler 來轉儲運行在系統上的所有作業的一些組成部分：

清單 12. 用來轉儲數據結構的修改過的 Kprope post_handler

```
1     void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags) {
2     struct task_struct *task;
3     read_lock(&tasklist_lock);
4     for_each_process(task) {
5     printk("pid =%x task-info_ptr=%lx\n", task->pid,
6     task->thread_info);
7     printk("thread-info element status=%lx,flags=%lx, cpu=%lx\n",
8     task->thread_info->status, task->thread_info->flags,
9     task->thread_info->cpu);
10    }
11    read_unlock(&tasklist_lock);
12    }
```

這個模塊應該插入到 do_fork 的偏移位置。

清單 13. pid 1508 和 1509 的結構體 thread_info 的輸出

```
1    $tail -10 /var/log/messages
2    Jun 22 18:14:25 llm05 kernel: thread-info element status=0,flags=0, cpu=1
3    Jun 22 18:14:25 llm05 kernel: pid =5e4 task-info_ptr=f5948000
4    Jun 22 18:14:25 llm05 kernel: thread-info element status=0,flags=8, cpu=0
5    Jun 22 18:14:25 llm05 kernel: pid =5e5 task-info_ptr=f5eca000
```

2.4 啟用奇妙的 SysRq 鍵

為了支持 SysRq 鍵，我們已經進行了編譯。這樣來啟用它：

```
1     $echo 1 > /proc/sys/kernel/sysrq
```

現在，您可以使用 Alt+SysRq+W 在控制台上或者到 /var/log/messages 中去查看所有插入的內核探測器。

清單 14. /var/log/messages 顯示出在 do_fork 插入了一個 Kprobe

```
1     Jun 23 10:24:48 linux-udp4749545uds kernel: SysRq : Show kprobes
2     Jun 23 10:24:48 linux-udp4749545uds kernel:
3     Jun 23 10:24:48 linux-udp4749545uds kernel: [<c011ea60>] do_fork+0x0/0x1de
```

3 使用 Kprobes 更好地進行調試

由於探測器事件處理器是作為系統斷點中斷處理器的擴展來運行，所以它們很少或者根本不依賴於系統 工具 —— 這樣可以被植入到大部分不友好的環境中（從中斷時間和任務時間到禁用的上下文間切換和支持 SMP 的代碼路徑）——都不會對系統性能帶來負面影響。

使用 Kprobes 的好處有很多。不需要重新編譯和重新引導內核就可以插入 printk。為了進行調試可以記錄 處理器寄存器的日誌，甚至進行修改 —— 不會干擾系統。類似地，同樣可以無干擾地記錄 Linux 內核數據結構的日誌，甚至進行修改。您甚至可以使用 Kprobes 調試 SMP 系統上的競態條件 —— 避免了您自己重新編譯和重新引導的所有 麻煩。您將發現內核調試比以往更為快速和簡單。

參考：

使用 Kprobes 調試內核