掌握 Linux 调试技术

在 Linux 上找出并解决程序错误的主要方法

您可以用各种方法来监控运行着的用户空间程序：可以为其运行调试器并单步调试该程序，添加打印语句，或者添加工具来分析程序。本文描述了几种可以用来调试在 Linux 上运行的程序的方法。我们将回顾四种调试问题的情况，这些问题包括段错误，内存溢出和泄漏，还有挂起。

Steve Best 目前在做 Linux 项目的日志纪录文件系统（Journaled File System，JFS）的工作。Steve 在操作系统方面有丰富的从业经验，他的着重的领域是文件系统、国际化和安全性。

2002 年 8 月 09 日

本文讨论了四种调试 Linux 程序的情况。在第 1 种情况中，我们使用了两个有内存分配问题的样本程序，使用 MEMWATCH 和 Yet Another Malloc Debugger（YAMD）工具来调试它们。在第 2 种情况中，我们使用了 Linux 中的 strace 实用程序，它能够跟踪系统调用和信号，从而找出程序发生错误的地方。在第 3 种情况中，我们使用 Linux 内核的 Oops 功能来解决程序的段错误，并向您展示如何设置内核源代码级调试器（kernel source level debugger，kgdb），以使用 GNU 调试器（GNU debugger，gdb）来解决相同的问题；kgdb 程序是使用串行连接的 Linux 内核远程 gdb。在第 4 种情况中，我们使用 Linux 上提供的魔术键控顺序（magic key sequence）来显示引发挂起问题的组件的信息。

常见调试方法

当您的程序中包含错误时，很可能在代码中某处有一个条件，您认为它为真（true），但实际上是假（false）。找出错误的过程也就是在找出错误后推翻以前一直确信为真的某个条件过程。

以下几个示例是您可能确信成立的条件的一些类型：

- 在源代码中的某处，某变量有特定的值。
- 在给定的地方，某个结构已被正确设置。
- 对于给定的 `if-then-else` 语句， `if` 部分就是被执行的路径。
- 当子例程被调用时，该例程正确地接收到了它的参数。

找出错误也就是要确定上述所有情况是否存在。如果您确信在子例程被调用时某变量应该有特定的值，那么就检查一下情况是否如此。如果您相信 `if` 结构会被执行，那么也检查一下情况是否如此。通常，您的假设都会是正确的，但最终您会找到与假设不符的情况。结果，您就会找出发生错误的地方。

调试是您无法逃避的任务。进行调试有很多种方法，比如将消息打印到屏幕上、使用调试器，或只是考虑程序执行的情况并仔细地揣摩问题所在。

在修正问题之前，您必须找出它的源头。举例来说，对于段错误，您需要了解段错误发生在代码的哪一行。一旦您发现了代码中出错的行，请确定该方法中变量的值、方法被调用的方式以及关于错误如何发生的详细情况。使用调试器将使找出所有这些信息变得很简单。如果没有调试器可用，您还可以使用其它的工具。（请注意，产品环境中可能并不提供调试器，而且 Linux 内核没有内建的调试器。）

本文将讨论一类通过人工检查代码不容易找到的问题，而且此类问题只在很少见的情况下存在。内存错误通常在多种情况同时存在时出现，而且您有时只能在部署程序之后才能发现内存错误。

实用的内存和内核工具

您可以使用 Linux 上的调试工具，通过各种方式跟踪用户空间和内核问题。请使用下面的工具和技术来构建和调试您的源代码：

用户空间工具：

- 内存工具：MEMWATCH 和 YAMD
- strace
- GNU 调试器（gdb）
- 魔术键控顺序

内核工具：

- 内核源代码级调试器（kgdb）
- 内建内核调试器（kdb）
- Oops

第 1 种情况：内存调试工具

C 语言作为 Linux 系统上标准的编程语言给予了我们对动态内存分配很大的控制权。然而，这种自由可能会导致严重的内存管理问题，而这些问题可能导致程序崩溃或随时间的推移导致性能降级。

内存泄漏（即 `malloc()` 内存在对应的 `free()` 调用执行后永不被释放）和缓冲区溢出（例如对以前分配到某数组的内存进行写操作）是一些常见的问题，它们可能很难检测到。这一部分将讨论几个调试工具，它们极大地简化了检测和找出内存问题的过程。

MEMWATCH

MEMWATCH 由 Johan Lindh 编写，是一个开放源代码 C 语言内存错误检测工具，您可以自己下载它（请参阅本文后面部分的 参考资料）。只要在代码中添加一个头文件并在 gcc 语句中定义了 MEMWATCH 之后，您就可以跟踪程序中的内存泄漏和错误了。MEMWATCH 支持 ANSI C，它提供结果日志纪录，能检测双重释放（double-free）、错误释放（erroneous free）、没有释放的内存（unfreed memory）、溢出和下溢等等。

清单 **1.** 内存样本（**test1.c**）

```
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"
int main(void)
{
  char *ptr1;
  char *ptr2;
  ptr1 = malloc(512);
  ptr2 = malloc(512);
  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
}
```

清单 1 中的代码将分配两个 512 字节的内存块，然后指向第一个内存块的指针被设定为指向第二个内存块。结果，第二个内存块的地址丢失，从而产生了内存泄漏。

现在我们编译清单 1 的 memwatch.c。下面是一个 makefile 示例：

test1

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

当您运行 test1 程序后，它会生成一个关于泄漏的内存的报告。清单 2 展示了示例 memwatch.log 输出文件。

清单 **2. test1 memwatch.log** 文件

```
  MEMWATCH 2.67 Copyright (C) 1992-1999 Johan Lindh
...
double-free: <4> test1.c(15), 0x80517b4 was freed from test1.c(14)
```

```
...
unfreed: <2> test1.c(11), 512 bytes at 0x80519e4
{FE FE FE FE FE FE FE FE FE FE FE FE ..............}
Memory usage statistics (global):
  N)umber of allocations made:  2
  L)argest memory usage :       1024
  T)otal of all alloc() calls:  1024
  U)nfreed bytes totals :       512
```

MEMWATCH 为您显示真正导致问题的行。如果您释放一个已经释放过的指针，它会告诉您。对于没有释放的内存也一样。日志结尾部分显示统计信息，包括泄漏了多少内存，使用了多少内存，以及总共分配了多少内存。

YAMD

YAMD 软件包由 Nate Eldredge 编写，可以查找 C 和 C++ 中动态的、与内存分配有关的问题。在撰写本文时，YAMD 的最新版本为 0.32。请下载 yamd-0.32.tar.gz（请参阅 参考资料）。执行 `make` 命令来构建程序；然后执行 `make install` 命令安装程序并设置工具。

一旦您下载了 YAMD 之后，请在 test1.c 上使用它。请删除 `#include memwatch.h` 并对 makefile 进行如下小小的修改：

使用 **YAMD** 的 **test1**

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

清单 3 展示了来自 test1 上的 YAMD 的输出。

清单 **3.** 使用 **YAMD** 的 **test1** 输出

```
YAMD version 0.32
Executable: /usr/src/test/yamd-0.32/test1
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal deallocation of this block
Address 0x40025e00, size 512
...
ERROR: Multiple freeing At
free of pointer already freed
Address 0x40025e00, size 512
...
WARNING: Memory leak
Address 0x40028e00, size 512
WARNING: Total memory leaks:
1 unfreed allocations totaling 512 bytes
*** Finished at Tue ... 10:07:15 2002
Allocated a grand total of 1024 bytes 2 allocations
Average of 512 bytes per allocation
Max bytes allocated at one time: 1024
24 K alloced internally / 12 K mapped now / 8 K max
Virtual program size is 1416 K
End.
```

YAMD 显示我们已经释放了内存，而且存在内存泄漏。让我们在清单 4 中另一个样本程序上试试 YAMD。

清单 **4.** 内存代码（**test2.c**）

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
  char *ptr1;
  char *ptr2;
  char *chptr;
  int i = 1;
  ptr1 = malloc(512);
  ptr2 = malloc(512);
  chptr = (char *)malloc(512);
  for (i; i <= 512; i++) {
    chptr[i] = 'S';
  }
  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
  free(chptr);
}
```

您可以使用下面的命令来启动 YAMD：

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

清单 5 显示了在样本程序 test2 上使用 YAMD 得到的输出。YAMD 告诉我们在 for 循环中有“越界（out-of-bounds）”的情况。

清单 **5.** 使用 **YAMD** 的 **test2** 输出

```
Running /usr/src/test/test2/test2
Temp output to /tmp/yamd-out.1243
*********
./run-yamd: line 101: 1248 Segmentation fault (core dumped)
YAMD version 0.32
Starting run: /usr/src/test/test2/test2
Executable: /usr/src/test/test2/test2
Virtual program size is 1380 K
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal allocation of this block
Address 0x4002be00, size 512
ERROR: Crash
...
Tried to write address 0x4002c000
Seems to be part of this block:
Address 0x4002be00, size 512
...
Address in question is at offset 512 (out of bounds)
Will dump core after checking heap.
Done.
```

MEMWATCH 和 YAMD 都是很有用的调试工具，它们的使用方法有所不同。对于 MEMWATCH，您需要添加包含文件 memwatch.h 并打开两个编译时间标记。对于链接（link）语句，YAMD 只需要 -g 选项。

Electric Fence

多数 Linux 分发版包含一个 Electric Fence 包，不过您也可以选择下载它。Electric Fence 是一个由 Bruce Perens 编写的 malloc() 调试库。它就在您分配内存后分配受保护的内存。如果存在 fencepost 错误（超过数组末尾运行），程序就会产生保护错误，并立即结束。通过结合 Electric Fence 和 gdb，您可以精确地跟踪到哪一行试图访问受保护内存。Electric Fence 的另一个功能就是能够检测内存泄漏。

第 **2** 种情况：使用 **strace**

strace 命令是一种强大的工具，它能够显示所有由用户空间程序发出的系统调用。strace 显示这些调用的参数并返回符号形式的值。strace 从内核接收信息，而且不需要以任何特殊的方式来构建内核。将跟踪信息发送到应用程序及内核开发者都很有用。在清单 6 中，分区的一种格式有错误，清单显示了 strace 的开头部分，内容是关于调出创建文件系统操作（ mkfs ）的。strace 确定哪个调用导致问题出现。

清单 **6. mkfs** 上 **strace** 的开头部分

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

清单 6 显示 ioctl 调用导致用来格式化分区的 mkfs 程序失败。 ioctl BLKGETSIZE64 失败。（ BLKGET-SIZE64 在调用 ioctl 的源代码中定义。) BLKGETSIZE64 ioctl 将被添加到 Linux 中所有的设备，而在这里，逻辑卷管理器还不支持它。因此，如果 BLKGETSIZE64 ioctl 调用失败，mkfs 代码将改为调用较早的 ioctl 调用；这使得 mkfs 适用于逻辑卷管理器。

第 **3** 种情况：使用 **gdb** 和 **Oops**

您可以从命令行使用 gdb 程序（Free Software Foundation 的调试器）来找出错误，也可以从诸如 Data Display Debugger（DDD）这样的几个图形工具之一使用 gdb 程序来找出错误。您可以使用 gdb 来调试用户空间程序或 Linux 内核。这一部分只讨论从命令行运行 gdb 的情况。

使用 `gdb program name` 命令启动 gdb。gdb 将载入可执行程序符号并显示输入提示符，让您可以开始使用调试器。您可以通过三种方式用 gdb 查看进程：

使用 attach 命令开始查看一个已经运行的进程；attach 将停止进程。

使用 run 命令执行程序并从头开始调试程序。

查看已有的核心文件来确定进程终止时的状态。要查看核心文件，请用下面的命令启动 gdb。 `gdb programname corefilename`
要用核心文件进行调试，您不仅需要程序的可执行文件和源文件，还需要核心文件本身。要用核心文件启动 gdb，请使用 -c 选项： `gdb -c core programname`

gdb 显示哪行代码导致程序发生核心转储。

在运行程序或连接到已经运行的程序之前，请列出您觉得有错误的源代码，设置断点，然后开始调试程序。您可以使用 `help` 命令查看全面的 gdb 在线帮助和详细的教程。

kgdb

kgdb 程序（使用 gdb 的远程主机 Linux 内核调试器）提供了一种使用 gdb 调试 Linux 内核的机制。kgdb 程序是内核的扩展，它让您能够在远程主机上运行 gdb 时连接到运行用 kgdb 扩展的内核机器。您可以接着深入到内核中、设置断点、检查数据并进行其它操作（类似于您在应用程序上使用 gdb 的方式）。这个补丁的主要特点之一就是运行 gdb 的主机在引导过程中连接到目标机器（运行要被调试的内核）。这让您能够尽早开始调试。请注意，补丁为 Linux 内核添加了功能，所以 gdb 可以用来调试 Linux 内核。

使用 kgdb 需要两台机器：一台是开发机器，另一台是测试机器。一条串行线（空调制解调器电缆）将通过机器的串口连接它们。您希望调试的内核在测试机器上运行；gdb 在开发机器上运行。gdb 使用串行线与您要调试的内核通信。

请遵循下面的步骤来设置 kgdb 调试环境：

1. 下载您的 Linux 内核版本适用的补丁。
2. 将组件构建到内核，因为这是使用 kgdb 最简单的方法。（请注意，有两种方法可以构建多数内核组件，比如作为模块或直接构建到内核中。举例来说，日志纪录文件系统（Journaled File System，JFS）可以作为模块构建，或直接构建到内核中。通过使用 gdb 补丁，我们就可以将 JFS 直接构建到内核中。）
3. 应用内核补丁并重新构建内核。
4. 创建一个名为 .gdbinit 的文件，并将其保存在内核源文件子目录中（换句话说就是 /usr/src/linux）。文件 .gdbinit 中有下面四行代码：

```
set remotebaud 115200

symbol-file vmlinux

target remote /dev/ttyS0

set output-radix 16
```

5. 将 append=gdb 这一行添加到 lilo，lilo 是用来在引导内核时选择使用哪个内核的引导载入程序。

```
image=/boot/bzImage-2.4.17

label=gdb2417

read-only
```

```
root=/dev/sda8
append="gdb gdbttyS=1 gdb-baud=115200 nmi_watchdog=0"
```

清单 7 是一个脚本示例，它将您在开发机器上构建的内核和模块引入测试机器。您需要修改下面几项：

`best@sfb`：用户标识和机器名。

`/usr/src/linux-2.4.17`：内核源代码树的目录。

`bzImage-2.4.17`：测试机器上将引导的内核名。

`rcp` 和 `rsync`：必须允许它在构建内核的机器上运行。

清单 **7.** 引入测试机器的内核和模块的脚本

```
set -x
rcp best@sfb: /usr/src/linux-2.4.17/arch/i386/boot/bzImage /boot/bzImage-2.4.17
rcp best@sfb:/usr/src/linux-2.4.17/System.map /boot/System.map-2.4.17
rm -rf /lib/modules/2.4.17
rsync -a best@sfb:/lib/modules/2.4.17 /lib/modules
chown -R root /lib/modules/2.4.17
lilo
```

现在我们可以通过改为使用内核源代码树开始的目录来启动开发机器上的 gdb 程序了。在本示例中，内核源代码树位于 /usr/src/linux-2.4.17。输入 gdb 启动程序。

如果一切正常，测试机器将在启动过程中停止。输入 gdb 命令 `cont` 以继续启动过程。一个常见的问题是，空调制解调器电缆可能会被连接到错误的串口。如果 gdb 不启动，将端口改为第二个串口，这会使 gdb 启动。

使用 **kgdb** 调试内核问题

清单 8 列出了 jfs_mount.c 文件的源代码中被修改过的代码，我们在代码中创建了一个空指针异常，从而使代码在第 109 行产生段错误。

清单 **8.** 修改过后的 **jfs_mount.c** 代码

```
int jfs_mount(struct super_block *sb)
{
...
int ptr;                        /* line 1 added */
jFYI(1, ("\nMount JFS\n"));
/ *
* read/validate superblock
* (initialize mount inode from the superblock)
* /
if ((rc = chkSuper(sb))) {
                goto errout20;
        }
108     ptr=0;                  /* line 2 added */
109     printk("%d\n",*ptr);    /* line 3 added */
```

清单 9 在向文件系统发出 mount 命令之后显示一个 gdb 异常。kgdb 提供了几条命令，如显示数据结构和变量值以及显示系统中的所有任务处于什么状态、它们驻留在何处、它们在哪些地方使用了 CPU 等等。清单 9 将显示回溯跟踪为该问题提供的信息； `where` 命令用来执行反跟踪，它将告诉被执行的调用在代码中的什么地方停止。

清单 **9. gdb** 异常和反跟踪

```
mount -t jfs /dev/sdb /jfs
Program received signal SIGSEGV, Segmentation fault.
jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
109             printk("%d\n",*ptr);
(gdb)where
#0 jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
#1 0xc01a0dbb in jfs_read_super ... at super.c:280
#2 0xc0149ff5 in get_sb_bdev ... at super.c:620
#3 0xc014a89f in do_kern_mount ... at super.c:849
#4 0xc0160e66 in do_add_mount ... at namespace.c:569
#5 0xc01610f4 in do_mount ... at namespace.c:683
#6 0xc01611ea in sys_mount ... at namespace.c:716
#7 0xc01074a7 in system_call () at af_packet.c:1891
#8 0x0 in -- ()
(gdb)
```

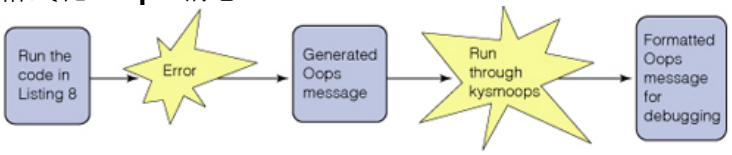
下一部分还将讨论这个相同的 JFS 段错误问题，但不设置调试器，如果您在非 kgdb 内核环境中执行清单 8 中的代码，那么它使用内核可能生成的 Oops 消息。

Oops 分析

Oops（也称 panic，慌张）消息包含系统错误的细节，如 CPU 寄存器的内容。在 Linux 中，调试系统崩溃的传统方法是分析在发生崩溃时发送到系统控制台的 Oops 消息。一旦您掌握了细节，就可以将消息发送到 ksymoops 实用程序，它将试图将代码转换为指令并将堆栈值映射到内核符号。在很多情况下，这些信息就足够您确定错误的可能原因是什么了。请注意，Oops 消息并不包括核心文件。

让我们假设系统刚刚创建了一条 Oops 消息。作为编写代码的人，您希望解决问题并确定什么导致了 Oops 消息的产生，或者您希望向显示了 Oops 消息的代码的开发者提供有关您的问题的大部分信息，从而及时地解决问题。Oops 消息是等式的一部分，但如果不通过 ksymoops 程序运行它也于事无补。下面的图显示了格式化 Oops 消息的过程。

格式化 **Oops** 消息



ksymoops 需要几项内容：Oops 消息输出、来自正在运行的内核的 System.map 文件，还有 /proc/ksyms、vmlinux 和 /proc/modules。关于如何使用 ksymoops，内核源代码 /usr/src/linux/Documentation/oops-tracing.txt 中或 ksymoops 手册页上有完整的说明可以参考。Ksymoops 反汇编代码部分，指出发生错误的指令，并显示一个跟踪部分表明代码如何被调用。

首先，将 Oops 消息保存在一个文件中以便通过 ksymoops 实用程序运行它。清单 10 显示了由安装 JFS 文件系统的 mount 命令创建的 Oops 消息，问题是由清单 8 中添加到 JFS 安装代码的那三行代码产生的。

清单 **10. ksymoops** 处理后的 **Oops** 消息

```
   ksymoops 2.4.0 on i686 2.4.17. Options used
... 15:59:37 sfb1 kernel: Unable to handle kernel NULL pointer dereference at
virtual address 0000000
... 15:59:37 sfb1 kernel: c01588fc
... 15:59:37 sfb1 kernel: *pde = 0000000
... 15:59:37 sfb1 kernel: Oops: 0000
... 15:59:37 sfb1 kernel: CPU:    0
... 15:59:37 sfb1 kernel: EIP:    0010:[jfs_mount+60/704]
... 15:59:37 sfb1 kernel: Call Trace: [jfs_read_super+287/688]
[get_sb_bdev+563/736] [do_kern_mount+189/336] [do_add_mount+35/208]
[do_page_fault+0/1264]
... 15:59:37 sfb1 kernel: Call Trace: [<c0155d4f>]...
... 15:59:37 sfb1 kernel: [<c0106e04 ...
... 15:59:37 sfb1 kernel: Code: 8b 2d 00 00 00 00 55 ...
>>EIP; c01588fc <jfs_mount+3c/2c0> <=====
...
Trace; c0106cf3 <system_call+33/40>
Code; c01588fc <jfs_mount+3c/2c0>
00000000 <_EIP>:
Code; c01588fc <jfs_mount+3c/2c0>  <=====
   0: 8b 2d 00 00 00 00         mov     0x0,%ebp     <=====
Code; c0158902 <jfs_mount+42/2c0>
   6:  55                       push    %ebp
```

接下来，您要确定 jfs_mount 中的哪一行代码引起了这个问题。Oops 消息告诉我们问题是由位于偏移地址 3c 的指令引起的。做这件事的办法之一是对 jfs_mount.o 文件使用 objdump 实用程序，然后查看偏移地址 3c。Objdump 用来反汇编模块函数，看看您的 C 源代码会产生什么汇编指令。清单 11 显示了使用 objdump 后您将看到的内容，接着，我们查看 jfs_mount 的 C 代码，可以看到空值是第 109 行引起的。偏移地址 3c 之所以很重要，是因为 Oops 消息将该处标识为引起问题的位置。

清单 **11. jfs_mount** 的汇编程序清单

```
  109   printk("%d\n",*ptr);
objdump jfs_mount.o
jfs_mount.o:    file format elf32-i386
Disassembly of section .text:
00000000 <jfs_mount>:
```

```
 0:55                        push %ebp
...
2c:   e8 cf 03 00 00     call    400 <chkSuper>
31:   89 c3                 mov     %eax,%ebx
33:   58                    pop     %eax
34:   85 db                 test    %ebx,%ebx
36:   0f 85 55 02 00 00 jne   291 <jfs_mount+0x291>
3c:   8b 2d 00 00 00 00 mov   0x0,%ebp << problem line above
42:   55                    push    %ebp
```

kdb

Linux 内核调试器（Linux kernel debugger，kdb）是 Linux 内核的补丁，它提供了一种在系统能运行时对内核内存和数据结构进行检查的办法。请注意，kdb 不需要两台机器，不过它也不允许您像 kgdb 那样进行源代码级别上的调试。您可以添加额外的命令，给出该数据结构的标识或地址，这些命令便可以格式化和显示基本的系统数据结构。目前的命令集允许您控制包括以下操作在内的内核操作：

处理器单步执行

执行到某条特定指令时停止

当存取（或修改）某个特定的虚拟内存位置时停止

当存取输入 / 输出地址空间中的寄存器时停止

对当前活动的任务和所有其它任务进行堆栈回溯跟踪（通过进程 ID）

对指令进行反汇编

追击内存溢出

您肯定不想陷入类似在几千次调用之后发生分配溢出这样的情形。

我们的小组花了许许多多时间来跟踪稀奇古怪的内存错误问题。应用程序在我们的开发工作站上能运行，但在新的产品工作站上，这个应用程序在调用 `malloc()` 两百万次之后就不能运行了。真正的问题是在大约一百万次调用之后发生了溢出。新系统之所有存在这个问题，是因为被保留的 `malloc()` 区域的布局有所不同，从而这些零散内存被放置在了不同的地方，在发生溢出时破坏了一些不同的内容。

我们用多种不同技术来解决这个问题，其中一种是使用调试器，另一种是在源代码中添加跟踪功能。在我职业生涯的大概也是这个时候，我便开始关注内存调试工具，希望能更快更有效地解决这些类型的问题。在开始一个新项目时，我最先做的事情之一就是运行 MEMWATCH 和 YAMD，看看它们是不是会指出内存管理方面的问题。

内存泄漏是应用程序中常见的问题，不过您可以使用本文所讲述的工具来解决这些问题。

第 4 种情况：使用魔术键控顺序进行回溯跟踪

如果在 Linux 挂起时您的键盘仍然能用，那请您使用以下方法来帮助解决挂起问题的根源。遵循这些步骤，您便可以显示当前运行的进程和所有使用魔术键控顺序的进程的回溯跟踪。

1. 您正在运行的内核必须是在启用 `CONFIG_MAGIC_SYS-REQ` 的情况下构建的。您还必须处在文本模式。CLTR+ALT+F1 会使您进入文本模式，CLTR+ALT+F7 会使您回到 X Windows。

2. 当在文本模式时，请按 <ALT+ScrollLock>，然后按 <Ctrl+ScrollLock>。上述魔术的击键会分别给出当前运行的进程和所有进程的堆栈跟踪。
3. 请查找 /var/log/messages。如果一切设置正确，则系统应该已经为您转换了内核的符号地址。回溯跟踪将被写到 /var/log/messages 文件中。

结束语

帮助调试 Linux 上的程序有许多不同的工具可供使用。 本文讲述的工具可以帮助您解决许多编码问题。能显示内存泄漏、溢出等等的位置的工具可以解决内存管理问题，我发现 MEMWATCH 和 YAMD 很有帮助。

使用 Linux 内核补丁会使 gdb 能在 Linux 内核上工作，这对解决我工作中使用的 Linux 的文件系统方面的问题很有帮助。此外，跟踪实用程序能帮助确定在系统调用期间文件系统实用程序什么地方出了故障。下次当您要摆平 Linux 中的错误时，请试试这些工具中的某一个。

参考资料

您可以参阅本文在 developerWorks 全球站点上的 英文原文.

下载 MEMWATCH。

请查看 Dynamic Probes 调试功能程序。

请阅读文章“ Linux software debugging with GDB”。（ *developerWorks*，2001 年 2 月）

请访问 IBM Linux Technology Center。

在 *developerWorks*Linux 专区可以找到 更多的 Linux 文章。