

developerWorks 中国 > 技术主题 > Linux > 文档库 >

技巧: 使用truss、strace或ltrace诊断软件的"疑难杂症"

本文通过三个实际案例演示如何使用truss、strace和ltrace这三个常用的调试工具来快速诊断软件的"疑难杂症"。

李凯斌 (pythonic@yeah.net), 项目经理

2004 年 12 月 01 日

0 评论

+ 内容

简介

进程无法启动, 软件运行速度突然变慢, 程序的"Segment Fault"等等都是让每个Unix系统用户头痛的问题, 本文通过三个实际案例演示如何使用truss、strace和ltrace这三个常用的调试工具来快速诊断软件的"疑难杂症"。

truss和strace用来跟踪一个进程的系统调用或信号产生的情况, 而ltrace用来跟踪进程调用库函数的情况。truss是早期为System V R4开发的调试程序, 包括Aix、FreeBSD在内的大部分Unix系统都自带了这个工具; 而strace最初是为SunOS系统编写的, ltrace最早出现在GNU/Debian Linux中。这两个工具现在也被移植到了大部分Unix系统中, 大多数Linux发行版都自带了strace和ltrace, 而FreeBSD也可通过Ports安装它们。

你不仅可以从命令行调试一个新开始的程序, 也可以把truss、strace或ltrace绑定到一个已有的PID上来调试一个正在运行的程序。三个调试工具的基本使用方法大体相同, 下面仅介绍三者共有, 而且是最常用的三个命令行参数:

```
-f :除了跟踪当前进程外, 还跟踪其子进程。
-o file :将输出信息写到文件file中, 而不是显示到标准错误输出(stderr)。
-p pid :绑定到一个由pid对应的正在运行的进程。此参数常用来调试后台进程。
```

使用上述三个参数基本上就可以完成大多数调试任务了, 下面举几个命令行例子:

```
truss -o ls.truss ls -al: 跟踪ls -al的运行, 将输出信息写到文件/tmp/ls.truss中。
strace -f -o vim.strace vim: 跟踪vim及其子进程的运行, 将输出信息写到文件vim.strace。
ltrace -p 234: 跟踪一个pid为234的已经在运行的进程。
```

三个调试工具的输出结果格式也很相似, 以strace为例:

```
brk(0) = 0x8062aa8
brk(0x8063000) = 0x8063000
mmap2(NULL, 4096, PROT_READ, MAP_PRIVATE, 3, 0x92f) = 0x40016000
```

每一行都是一条系统调用, 等号左边是系统调用的函数名及其参数, 右边是该调用的返回值。truss、strace和ltrace的工作原理大同小异, 都是使用ptrace系统调用跟踪调试运行中的进程, 详细原理不在本文讨论范围内, 有兴趣可以参考它们的源代码。

下面举两个实例演示如何利用这三个调试工具诊断软件的"疑难杂症":

↑ 回页首

案例一: 运行clint出现Segment Fault错误

操作系统: FreeBSD-5.2.1-release

clint是一个C++静态源代码分析工具, 通过Ports安装好之后, 运行:

```
# clint foo.cpp
Segmentation fault (core dumped)
```

在Unix系统中遇见"Segmentation Fault"就像在MS Windows中弹出"非法操作"对话框一样令人讨厌。OK, 我们用truss给clint"把把脉":

```
# truss -f -o clint.truss clint
Segmentation fault (core dumped)
# tail clint.truss
739: read(0x6,0x806f000,0x1000) = 4096 (0x1000)
739: fstat(6,0xbfbfe4d0) = 0 (0x0)
739: fcntl(0x6,0x3,0x0) = 4 (0x4)
739: fcntl(0x6,0x4,0x0) = 0 (0x0)
739: close(6) = 0 (0x0)
739: stat("/root/.clint/plugins",0xbfbfe680) = ERR#2 'No such file or directory'
SIGNAL 11
SIGNAL 11
Process stopped because of: 16
process exit, rval = 139
```



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

开始您的试用

我们用truss跟踪clint的系统调用执行情况, 并把结果输出到文件clint.truss, 然后用tail查看最后几行。注意看clint执行的最后一条系统调用(倒数第五行): stat("/root/.clint/plugins",0xbfbfe680) ERR#2 'No such file or directory', 问题就出在这里:clint找不到目录"/root/.clint/plugins", 从而引发了段错误。怎样解决?很简单: mkdir -p /root/.clint/plugins, 不过这次运行clint还是会"Segmentation Fault"9。继续用truss跟踪, 发现clint还需要这个目录"/root/.clint/plugins/python", 建好这个目录后clint终于能够正常运行了。

[↑ 回页首](#)

案例二: vim启动速度明显变慢

操作系统: FreeBSD-5.2.1-release

vim版本为6.2.154, 从命令行运行vim后, 要等待近半分钟才能进入编辑界面, 而且没有任何错误输出。仔细检查了.vimrc和所有的vim脚本都没有错误配置, 在网上也找不到类似问题的解决办法, 难不成要hacking source code? 没有必要, 用truss就能找到问题所在:

```
# truss -f -D -o vim.truss vim
```

这里-D参数的作用是: 在每行输出前加上相对时间戳, 即每执行一条系统调用所耗费的时间。我们只要关注哪些系统调用耗费的时间比较长就可以了, 用less仔细查看输出文件vim.truss, 很快就找到了疑点:

```
735: 0.000021511 socket(0x2,0x1,0x0) = 4 (0x4)
735: 0.000014248 setsockopt(0x4,0x6,0x1,0xbfbfe3c8,0x4) = 0 (0x0)
735: 0.000013688 setsockopt(0x4,0xffff,0x8,0xbfbfe2ec,0x4) = 0 (0x0)
735: 0.000203657 connect(0x4,{ AF_INET 10.57.18.27:6000 },16) ERR#61 'Connection refused'
735: 0.000017042 close(4) = 0 (0x0)
735: 1.009366553 nanosleep(0xbfbfe468,0xbfbfe460) = 0 (0x0)
735: 0.000019556 socket(0x2,0x1,0x0) = 4 (0x4)
735: 0.000013409 setsockopt(0x4,0x6,0x1,0xbfbfe3c8,0x4) = 0 (0x0)
735: 0.000013130 setsockopt(0x4,0xffff,0x8,0xbfbfe2ec,0x4) = 0 (0x0)
735: 0.000272102 connect(0x4,{ AF_INET 10.57.18.27:6000 },16) ERR#61 'Connection refused'
735: 0.000015924 close(4) = 0 (0x0)
735: 1.009338338 nanosleep(0xbfbfe468,0xbfbfe460) = 0 (0x0)
```

vim试图连接10.57.18.27这台主机的6000端口(第四行的connect()), 连接失败后, 睡眠一秒钟继续重试(第6行的nanosleep())。以上片断循环出现了十几次, 每次都要耗费一秒多钟的时间, 这就是vim明显变慢的原因。可是, 你肯定会纳闷: "vim怎么会无缘无故连接其它计算机的6000端口呢?". 问得好, 那么请你回想一下6000是什么服务的端口? 没错, 就是X Server。看来vim是要把输出定向到一个远程X Server, 那么Shell中肯定定义了DISPLAY变量, 查看.cshrc, 果然有这么一行: setenv DISPLAY \${REMOTEHOST}:0, 把它注释掉, 再重新登录, 问题就解决了。

[↑ 回页首](#)

案例三: 用调试工具掌握软件的工作原理

操作系统: Red Hat Linux 9.0

用调试工具实时跟踪软件的运行情况不仅是诊断软件"疑难杂症"的有效的手段, 也可帮助我们理清软件的"脉络", 即快速掌握软件的运行流程和工作原理, 不失为一种学习源代码的辅助方法。下面这个案例展现了如何使用strace通过跟踪别的软件来"触发灵感", 从而解决软件开发中的难题的。

大家都知道, 在进程内打开一个文件, 都有唯一一个文件描述符(fd: file descriptor)与这个文件对应。而本人在开发一个软件过程中遇到这样一个问题: 已知一个fd, 如何获取这个fd所对应文件的完整路径? 不管是Linux、FreeBSD或是其它Unix系统都没有提供这样的API, 怎么办呢? 我们换个角度思考: Unix下有没有什么软件可以获取进程打开了哪些文件? 如果你经验足够丰富, 很容易想到lsdf, 使用它既可以知道进程打开了哪些文件, 也可以了解一个文件被哪个进程打开。

好, 我们用一个小程序来试验一下lsdf, 看它是如何获取进程打开了哪些文件。

```
/* testlsdf.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(void)
{
    open("/tmp/foo", O_CREAT|O_RDONLY); /* 打开文件/tmp/foo */
    sleep(1200); /* 睡眠1200秒, 以便进行后续操作 */
    return 0;
}
```

将testlsdf放入后台运行, 其pid为3125。命令lsdf -p 3125查看进程3125打开了哪些文件, 我们用strace跟踪lsdf的运行, 输出结果保存在lsdf.strace中:

```
# gcc testlsdf.c -o testlsdf
# ./testlsdf &
[1] 3125
# strace -o lsdf.strace lsdf -p 3125
```

我们以"/tmp/foo"为关键字搜索输出文件lsdf.strace, 结果只有一条:

```
# grep '/tmp/foo' lsdf.strace
readlink("/proc/3125/fd/3", "/tmp/foo", 4096) = 8
```

原来lsyf巧妙的利用了/proc/nnnn/fd/目录(nnnn为pid):Linux内核会为每一个进程在/proc/建立一个以其pid为名的目录用来保存进程的相关信息,而其子目录fd保存的是该进程打开的所有文件的fd。目标离我们很近了。好,我们到/proc/3125/fd/看个究竟:

```
# cd /proc/3125/fd/
# ls -l
total 0
lrwx----- 1 root    root      64 Nov  5 09:50 0 -> /dev/pts/0
lrwx----- 1 root    root      64 Nov  5 09:50 1 -> /dev/pts/0
lrwx----- 1 root    root      64 Nov  5 09:50 2 -> /dev/pts/0
lr-x----- 1 root    root      64 Nov  5 09:50 3 -> /tmp/foo
# readlink /proc/3125/fd/3
/tmp/foo
```

答案已经很明显了:/proc/nnnn/fd/目录下的每一个fd文件都是符号链接,而此链接就指向被该进程打开的一个文件。我们只要用readlink()系统调用就可以获取某个fd对应的文件了,代码如下:

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
int get_pathname_from_fd(int fd, char pathname[], int n)
{
    char buf[1024];
    pid_t pid;
    bzero(buf, 1024);
    pid = getpid();
    snprintf(buf, 1024, "/proc/%i/fd/%i", pid, fd);
    return readlink(buf, pathname, n);
}
int main(void)
{
    int fd;
    char pathname[4096];
    bzero(pathname, 4096);
    fd = open("/tmp/foo", O_CREAT|O_RDONLY);
    get_pathname_from_fd(fd, pathname, 4096);
    printf("fd=%d; pathname=%s\n", fd, pathname);
    return 0;
}
```

【注】出于安全方面的考虑,在FreeBSD 5 之后系统默认已经不再自动装载proc文件系统,因此,要想使用truss或strace跟踪程序,你必须手工装载proc文件系统:mount -t procfs proc /proc; 或者在/etc/fstab中加上一行:

proc	/proc	procfs	rw	0	0
------	-------	--------	----	---	---

ltrace不需要使用procfs。

参考资料

- truss(1) manual page
- strace(1) manual page
- ltrace(1) manual page
- ptrace(2) manual page
- lsof(1) manual page
- Debugging with strace: <http://www.devchannel.org/devtoolschannel/03/10/24/2057246.shtml>



IBM Bluemix 资源中心
文章、教程、演示,帮助您构建、部署和管理云应用。



developerWorks 中文社区
立即加入来自 IBM 的专业 IT 社交网络。



Bluemixathon 挑战赛
为灾难恢复构建应用,赢取现金大奖。

条评论

请 [登录](#) 或 [注册](#) 后发表评论。

添加评论:

注意:评论中不支持 HTML 语法

☐ 有新评论时提醒我

剩余 1000 字符

发布


快来添加第一条评论

[帮助](#)

[联系编辑](#)

[提交内容](#)

[订阅源](#)

 [新浪微博](#)

[报告滥用](#)

[使用条款](#)

[第三方提示](#)

[隐私条约](#)

[浏览辅助](#)

[IBM教育学院教育培养计划](#)

[IBM创业企业全球扶持计划](#)

[ISV 资源 \(英语\)](#)

[dW 中国每周时事通讯](#)

[选择语言:](#)

[English](#)

[中文](#)

[日本語](#)

[Русский](#)

[Português \(Brasil\)](#)

[Español](#)

[Việt](#)

