

Monitoring Oracle I/O by using sytemtap

Problem

For a long time, it was very hard for a DBA (or a system administrator, for that matter) to answer a very simple question: who is generating all that I/O on my system? In other words: which process is generating most of the I/O requests? On Solaris, there is a tool called "dtrace" which can answer that question. Bad news is that dtrace is only available on Solaris 10. Most of my production SUN boxes are still running Solaris 8 and 9 and CIO has made a decision, based on price, of course, to switch to Linux. That means that "dtrace" is out of the question and that a Linux solution must be found.

Solution

What is available on Linux? Well, as it turns out, there is a worthy alternative, not as good or fancy as dtrace, but good enough to be very, very useful. On kernels 2.6 and newer (Red Hat EL 4.0 and 5.0) there is a tool called "systemtap", open source software, available on <http://sourceware.org/systemtap/> which does pretty much the same thing, although not as elegantly as "dtrace". So, what does "systemtap" do? Systemtap is a programming language in itself, used to intercept system calls. This little language has all the familiar constructs: variables, loops and conditional statements. Syntax is rather simple and described in detail in the manual pages and online documentation. I will not delve into the language itself, as anybody even so slightly familiar with the PL/SQL will immediately recognize all of the constructs. Systemtap intecepts the calls to the kernel by translating the "systemtap" script into C, compiling it and inserting it into the running kernel as a module. The inserted module is visible through lsmod:

```
[root@medo tmp]# lsmod
Module                Size  Used by
stap_8900df1b1824fed402d0081a17801475_760 51632 1
autofs4               24261  2
ipv6                  275969  26
vfat                  16193  1
fat                   53469  1 vfat
dm_multipath         21833  0
video                 19909  0
sbs                   19073  0
i2c_ec                9025  1 sbs
dock                  13977  0
button                11857  0
battery               13893  0
asus_acpi              20445  0
backlight             10177  1 asus_acpi
ac                     9157  0
lp                    16265  0
snd_ens1371           29281  0
snd_ac97_codec        93413  1 snd_ens1371
ac97_bus              6465  1 snd_ac97_codec
snd_seq_dummy         7877  0
snd_seq_oss           33601  0
snd_seq_midi_event    11201  1 snd_seq_oss
snd_seq               51249  5 snd_seq_dummy,snd_seq_oss,snd_seq_midi_event
snd_pcm_oss           43617  0
snd_mixer_oss         19393  1 snd_pcm_oss
snd_mpu401            12393  0
```

```

snd_mpu401_uart    12609  1 snd_mpu401
snd_pcm            75205  3 snd_ens1371,snd_ac97_codec,snd_pcm_oss
snd_rawmidi       26817  2 snd_ens1371,snd_mpu401_uart
snd_seq_device    11853  4 snd_seq_dummy,snd_seq_oss,snd_seq,snd_rawmidi
floppy            58789  0
snd_timer         25157  2 snd_seq,snd_pcm
ide_cd            40673  0
snd               53829  12
snd_ens1371,snd_ac97_codec,snd_seq_oss,snd_seq,snd_pcm_oss,snd_mixer_oss,snd_mpu401,snd_mpu401_uart,snd_pcm,snd_rawmidi,snd_seq_device,snd_timer
nvidia            6844084 22
pcspkr            7233  0
snd_page_alloc    13769  1 snd_pcm
tulip              53089  0
i2c_nforce2       9793  0
cdrom              37345  1 ide_cd
parport_pc        30053  1
soundcore         11681  1 snd
ns558             9153  0
gameport          19401  3 snd_ens1371,ns558
parport           38537  2 lp,parport_pc
serio_raw         10821  0
i2c_core          24897  3 i2c_ec,nvidia,i2c_nforce2
dm_snapshot       20849  0
dm_zero           6209  0
dm_mirror         25301  0
dm_mod            58253  9 dm_multipath,dm_snapshot,dm_zero,dm_mirror
sata_nv           22469  0
libata            104661  1 sata_nv
sd_mod            24129  0
scsi_mod          138093  2 libata,sd_mod
ext3              125641  9
jbd               60777  1 ext3
ehci_hcd          34125  0
ohci_hcd          23493  0
uhci_hcd          26705  0
[root@medo tmp]#

```

The first module is a module with a monstrous name which looks like this:

```
stap_8900df1b1824fed402d0081a17801475_760
```

This module intercepts the system services and kernel functions and performs what it's programmed to do. In the manual page, available at the same site as the software itself, the term "probe" is used rather than the term "intercept". Of course, creating and inserting modules on the fly is a highly privileged operation which requires root access. That is the only problem with stap: it requires root access, but that is the nature of the beast.

How do "stap" scripts look? Without further ado, here is the first useful script:

```

#!/usr/bin/stap
#
# This script continuously lists the top 20 oracle processes on the system
# sorted by reads. This script expects uid() of the user oracle as an
# argument.

global ioreqs
global oid=$1

```

```

function print_top () {
    cnt=0
    log ("PROC\t\t\tREADS")
    foreach ([proc] in ioreqs-) {
        printf("%-10d\t\t%5d\n",proc, ioreqs[proc])
        if (cnt++ == 20)
            break
    }
    printf("-----\n")
    delete ioreqs
}

probe syscall.read {
    if (uid()==oid)
        ioreqs[pid()]++
}

# print top ioreqs every 5 seconds
probe timer.ms(5000) {
    print_top ()
}

```

The output, primitive as it is, looks like this:

```

[root@medo tmp]# ./ora_reads 501
PROC      READS
4681         7
4691         4
4697         1
-----
PROC      READS
4681        14
4701         8
4691         4
4697         1
-----
PROC      READS
4681        14
4691         2
4697         1
-----

```

Of course, the argument 501 is the user id of "oracle" on my system:

```

$ id
uid=501(oracle) gid=501(dba) groups=501(dba)

```

Script must be executed as "root" and interrupted by SIGINT (usually, Ctrl-C). Now, we can take a look what process is the process 4681:

```

$ ps -fp 4681
UID    PID  PPID  C  STIME TTY      TIME CMD
oracle 4681  1    0 00:05 ?        00:00:00 ora_pmon_10G
$

```

It's PMON, the process that is expected to be I/O intensive from time to time. I don't have any useful application system running on my PC, so this should suffice for now. Unfortunately, systemtap will not work on RH 2.1 or 3.0 and it will require an additional installation of debug information for the kernel. It is, however, possible to do this with a supported kernel from RH, without downloading from the kernel.org and creating a custom kernel just to run it, as is necessary with atop. Of course, formatting the output is something that still requires pathologically eclectic rubbish lister, my favorite tool.

One thing must be noted here: systemtap can not yet provide the length of the data that was read, only the number of times that "read" system service was called. Strictly speaking, we can only see which process has done the most calls to "read", but we cannot answer the question about the process which has read the most data. We can, however, answer both questions for the network calls.

Why did I chose to intercept reads, rather than writes? By the very nature of the Oracle RDBMS, most of the writing is done by the database writer, the DBWR process or processes. Intercepting writes would merely get me the usual culprits, namely DBWR, LGWR and CKPT processes. It is much harder to see which processes are performing the largest number of reads at the moment. That is why the reads were chosen over the writes. It is also possible to trace kernel functions, like "vfs_read", in which case the script will only list the processes employing "vfs_read" function, a generalized "read" function used by the kernel to call "read" function for the underlying file system.

Monitoring I/O is useful, but is not the only thing in Oracle RDBMS that stap can be used for. Another interesting use of stap is network accounting. Here is an interesting way to find processes which have sent or received the most of data in the last 5 seconds:

```
#!/usr/bin/stap
global netreqs
global oid=$1

function print_top () {
  cnt=0
  log ("PROC\t\t\tNet Traffic")
  foreach ([proc] in netreqs-) {
    printf("%-10d\t\t%5d\n",proc, netreqs[proc])
    if (cnt++ == 20)
      break
  }
  printf("-----\n")
  delete netreqs
}

probe netdev.receive,netdev.transmit {
  if (uid()==oid)
    netreqs[pid()]=netreqs[pid()]+length
}

# print top ioreqs every 5 seconds
probe timer.ms(5000) {
  print_top ()
}
```

The main probe here is the one probing for the amount of received or sent data. If the process belongs to the user specified by the first command line argument, the length of the communicated data will be accounted for. The output is here:

PROC	Net Traffic
3780	589186
2916	3116
3030	1558
2994	1492

Process 3780 is a web browser, which naturally communicates over the network:

```
$ ps -fp 3780
UID      PID  PPID  C STIME TTY          TIME CMD
mgogala  3780 3775 23 21:10 ?           00:04:14 /usr/lib/firefox-1.5.0.12/firefox
```

To make a long story short, systemtap is an exciting new monitoring tool. It will probably become fully usable only with Red Hat Linux 5.0 and later (I am testing this on Fedora Core 6) but the promise is certainly here. Systemtap can intercept and account for all types of system services and can be used to monitor interaction between Oracle RDBMS and the operating system. At this stage, systemtap produces only ASCII output but that can be post-processed by any number of tools, ranging from programming languages like Perl and PHP, word processors like MS-Word or OpenOffice to Oracle external tables. The tool itself is still very new and not very popular but it is already able to provide some really important information. I expect it to become much more useful in not so distant future. It will probably be frequently used in conjunction with Oracle 11R2.