開
放
的
열린
مفتوح
libre
मुक्त
ಮುಕ್ತ
livre
libero
ముక్త
开放的
açık
open
nyílt
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை

open

USE IMPROVE EVANGELIZE

OpenSolaris:
Dynamic Tracing: DTrace
(abbreviated from Jim Mauro's Usenix preso)

Harry J Foxwell, PhD
Principal Consultant
Oracle Corporation

harry.foxwell@oracle.com

Acknowledgements

- Some of this material represents an aggregation and consolidation of existing material, including the original **Usenix** paper*. A significant amount of the material contained in these slides was inserted from the slide sets, blogs, emails, letters, post cards, faxes, telegrams and sticky notes of others:

- Stefan Parvu
- Brendan Gregg
- Bryan Cantrill
- Mike Shapiro
- Adam Leventhal
- Jon Haslam
- Bart Smaalders
- Jarod Jenson
- Chad Mynhier
- Jim Fiori
- Jonathan Adams
- John Birrell
- Simon Ritter
- Angelo Rajadurai
- Bob Netherton
- Peter Karlsson
- Roch Bourbonnais
- Richard McDougall
- Keith McGuigan
- John Levon
- Chip Bennett
- **Jim Mauro**

***http://www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf**

Resources

- DTrace documentation

  http://wikis.sun.com/display/DTrace/Documentation
- DTrace tutorials, scripts, etc

  http://www.solarisinternals.com/wiki/index.php/DTrace_Topics
- DTrace community (lots 'o stuff)

  http://www.opensolaris.org/os/community/dtrace/
- DTrace ToolKit

  http://www.brendangregg.com/dtrace.html#DTraceToolkit
- Blogs, blogs, blogs

  http://blogs.sun.com

Scripts!

- Sample commands and scripts

/usr/demo/dtrace/*.d (on Solaris 10 and OpenSolaris)

http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_One_Liners

http://www.nbl.fi/~nbl97/solaris/dtrace/index.html

http://www.solarisinternals.com/si/dtrace/index.php

http://www.opensolaris.org/os/community/dtrace/dtracetoolkit/

What is DTrace?

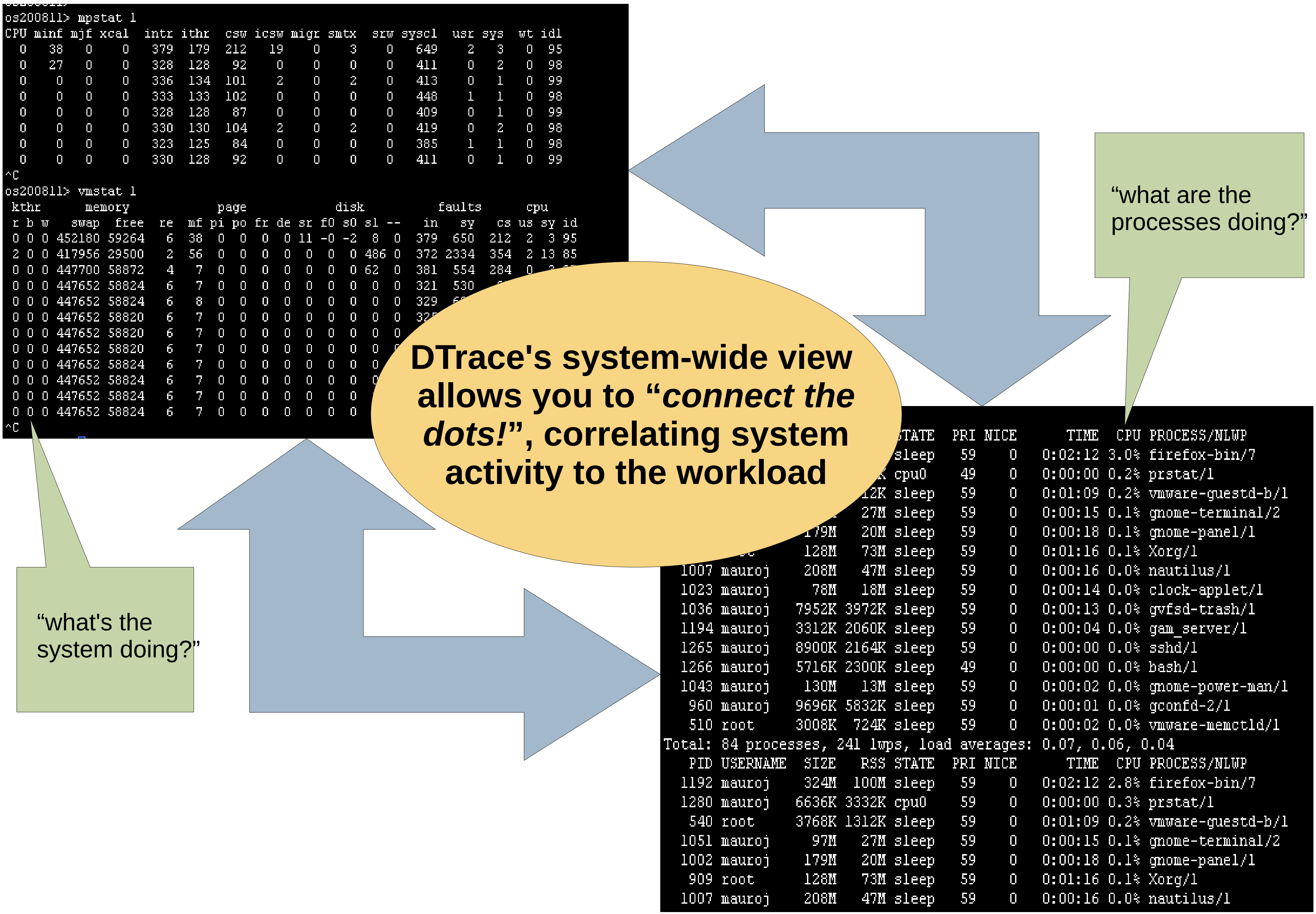
- DTrace is a facility for the **dynamic** instrumentation of **production** systems, for the purpose of troubleshooting and analysis.
  - First introduced in Solaris 10 (3/05)
  - Ported to **Mac OS X** and FreeBSD
  - but **not** Linux (See **SystemTap** instead)
- DTrace is many things, in particular:
  - An **instrumentation** framework
  - A **programming language**
- DTrace provides **observability** across the entire software stack from **one tool**. This allows you to examine software execution like never before.
  - Instrument kernel and user software in a unified fashion

System Analysis

- Traditional development and debugging tools are tightly bound to the language and/or development framework
  - SunStudio, IDE tools, etc
  - Lack **system view** process only or kernel only)
- System tools lack correlation to the workload
  - sar, mpstat, vmstat, iostat, etc
  - You can see what the system is doing, but...
- Hard to debug *transient* problems with truss(1), pstack(1), prstat(1M), etc
- Only mdb(1) designed for systemic problems, but intended for postmortem analysis
  - mdb(1) is useful for some live system views

DTrace

- A powerful framework for real-time analysis and observability. System **and** process centric
- Dynamic instrumentation of the kernel **and** applications
- Dynamically interpreted language allows for arbitrary actions and predicates in multiple points of instrumentation
- **Designed for live production systems**:
  - a **totally safe** way to inspect **live data** on **production systems**



```
os200811> mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   38   0    0   379  179  212   19    0    3    0   649    2   3   0  95
  0   27   0    0   328  128   92    0    0    0    0   411    0   2   0  98
  0    0   0    0   336  134  101    2    0    2    0   413    0   1   0  99
  0    0   0    0   333  133  102    0    0    0    0   448    1   1   0  98
  0    0   0    0   328  128   87    0    0    0    0   409    0   1   0  99
  0    0   0    0   330  130  104    2    0    2    0   419    0   2   0  98
  0    0   0    0   323  125   84    0    0    0    0   385    1   1   0  98
  0    0   0    0   330  128   92    0    0    0    0   411    0   1   0  99
^C
os200811> vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr f0 s0 s1 --   in   sy   cs us sy id
 0 0 0 452180 59264   6  38  0  0  0  0 11 -0 -2  8  0  379  650  212  2  3 95
 2 0 0 417956 29500   2  56  0  0  0  0  0  0  0 486 0  372 2334  354  2 13 85
 0 0 0 447700 58872   4   7  0  0  0  0  0  0  0 62  0  381  554  284  0
 0 0 0 447652 58824   6   7  0  0  0  0  0  0  0  0  0  321  530
 0 0 0 447652 58824   6   8  0  0  0  0  0  0  0  0  0  329
 0 0 0 447652 58820   6   7  0  0  0  0  0  0  0  0  0
 0 0 0 447652 58820   6   7  0  0  0  0  0  0  0  0  0
 0 0 0 447652 58820   6   7  0  0  0  0  0  0  0  0
 0 0 0 447652 58824   6   7  0  0  0  0  0  0  0  0
 0 0 0 447652 58824   6   7  0  0  0  0  0  0  0
 0 0 0 447652 58824   6   7  0  0  0  0  0  0  0
 0 0 0 447652 58824   6   7  0  0  0  0  0  0  0
^C
```

“what's the system doing?”

DTrace's system-wide view allows you to “*connect the dots!*”, correlating system activity to the workload

“what are the processes doing?”

```
                          STATE  PRI NICE     TIME  CPU PROCESS/NLWP
                          sleep   59    0  0:02:12 3.0% firefox-bin/7
                          cpu0    49    0  0:00:00 0.2% prstat/1
                      12K sleep   59    0  0:01:09 0.2% vmware-guestd-b/1
                      27M sleep   59    0  0:00:15 0.1% gnome-terminal/2
                179M  20M sleep   59    0  0:00:18 0.1% gnome-panel/1
                128M  73M sleep   59    0  0:01:16 0.1% Xorg/1
  1007 mauroj   208M  47M sleep   59    0  0:00:16 0.0% nautilus/1
  1023 mauroj    78M  18M sleep   59    0  0:00:14 0.0% clock-applet/1
  1036 mauroj  7952K 3972K sleep  59    0  0:00:13 0.0% gvfsd-trash/1
  1194 mauroj  3312K 2060K sleep  59    0  0:00:04 0.0% gam_server/1
  1265 mauroj  8900K 2164K sleep  59    0  0:00:00 0.0% sshd/1
  1266 mauroj  5716K 2300K sleep  49    0  0:00:00 0.0% bash/1
  1043 mauroj   130M  13M sleep   59    0  0:00:02 0.0% gnome-power-man/1
   960 mauroj  9696K 5832K sleep  59    0  0:00:01 0.0% gconfd-2/1
   510 root    3008K  724K sleep  59    0  0:00:02 0.0% vmware-memctld/1
Total: 84 processes, 241 lwps, load averages: 0.07, 0.06, 0.04
   PID USERNAME  SIZE   RSS STATE  PRI NICE     TIME  CPU PROCESS/NLWP
  1192 mauroj   324M 100M sleep   59    0  0:02:12 2.8% firefox-bin/7
  1280 mauroj  6636K 3332K cpu0   59    0  0:00:00 0.3% prstat/1
   540 root    3768K 1312K sleep  59    0  0:01:09 0.2% vmware-guestd-b/1
  1051 mauroj    97M  27M sleep   59    0  0:00:15 0.1% gnome-terminal/2
  1002 mauroj   179M  20M sleep   59    0  0:00:18 0.1% gnome-panel/1
   909 root     128M  73M sleep   59    0  0:01:16 0.1% Xorg/1
  1007 mauroj   208M  47M sleep   59    0  0:00:16 0.0% nautilus/1
```

DTrace

An Observability Revolution

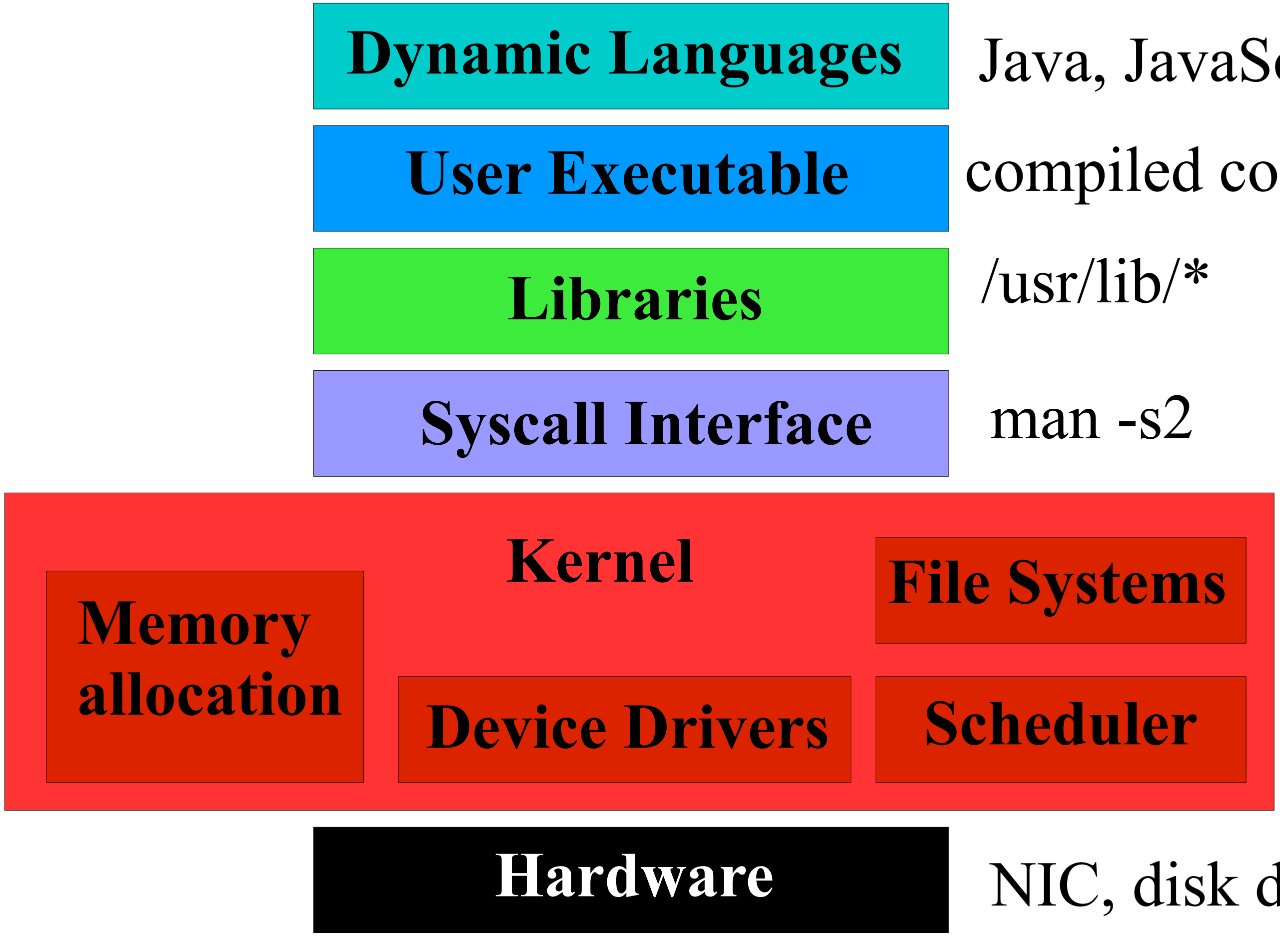
- Ease-of-use and instant gratification engenders serious hypothesis testing
- Instrumentation directed by high-level control language (not unlike AWK or C) for easy scripting and command line use
  - Build your DTrace toolbox
- Comprehensive probe coverage and powerful data management allow for concise answers to arbitrary questions
  - *What are these system calls, and who's executing them?*

DTrace

- Safe and comprehensive: tens-of-thousands of data monitoring points (dtrace -l)
  - Inspect kernel and user space
- Reduced costs: problems usually found in minutes or hours, not days or weeks
- Flexibility: DTrace lets you create your own custom programs to dynamically instrument the system
- No need to instrument your applications via source code modifications; no need to stop or restart them

The Entire Software Stack

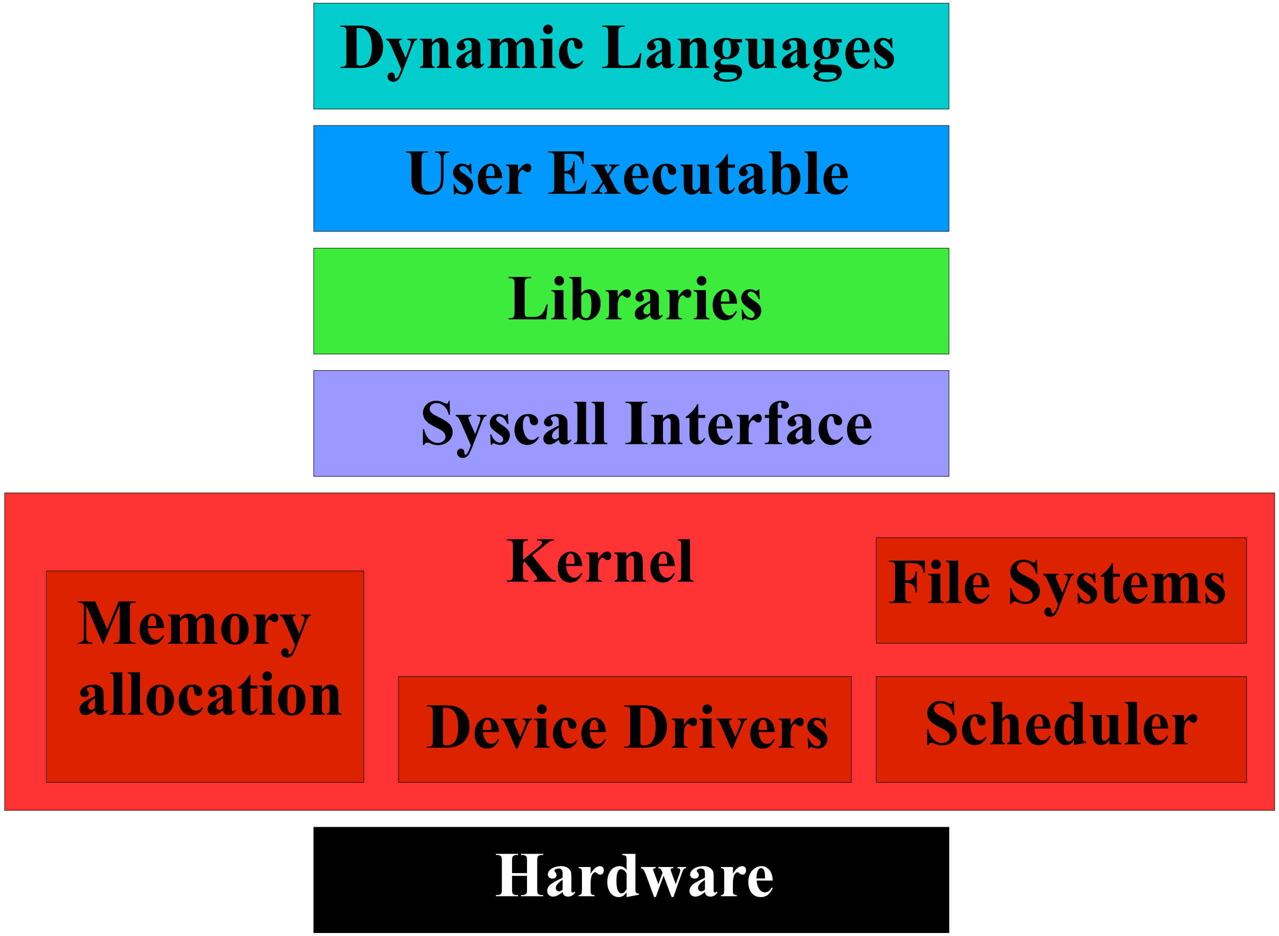
- How did you analyze these?



	Examples:
Dynamic Languages	Java, JavaScript, ...
User Executable	compiled code, /usr/bin/*
Libraries	/usr/lib/*
Syscall Interface	man -s2
Kernel **Memory allocation** **File Systems** **Device Drivers** **Scheduler**	VFS, DNLC, UFS, ZFS, TCP, IP, ... sd, st, hme, eri, ...
Hardware	NIC, disk data controller, CPU

The Entire Software Stack

- It was possible, but difficult.



Dynamic Languages

User Executable

Libraries

Syscall Interface

Kernel

Memory allocation

File Systems

Device Drivers

Scheduler

Hardware

Previously:

debuggers

truss -ua.out

apptrace, sotruss

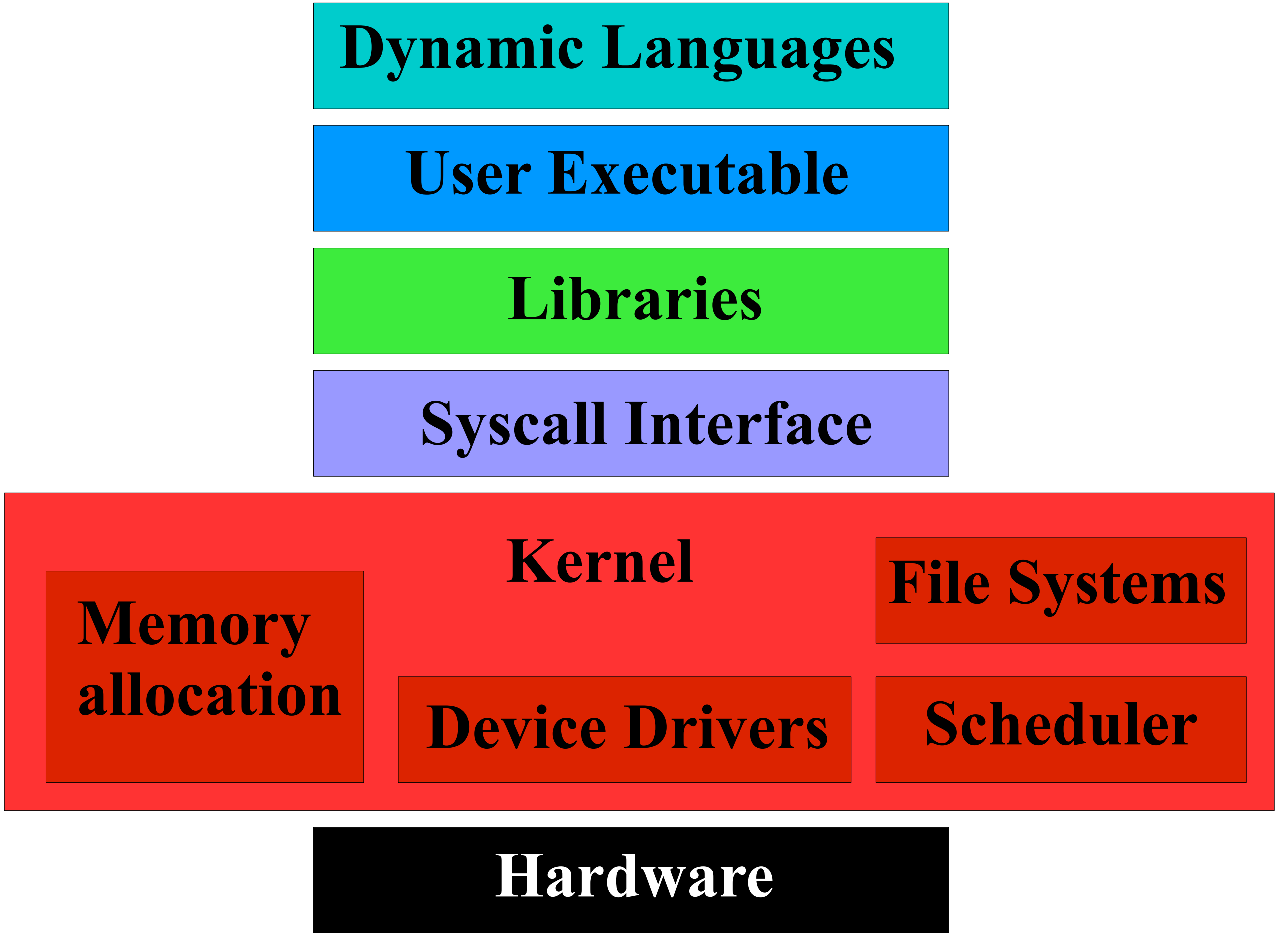
truss

prex; tnf*
lockstat
mdb

kstat, PICs, guesswork

The Entire Software Stack

- DTrace is all seeing:



	DTrace visibility:
Dynamic Languages	Yes, with providers
User Executable	Yes
Libraries	Yes
Syscall Interface	Yes
Kernel **Memory allocation** **File Systems** **Device Drivers** **Scheduler**	Yes
Hardware	Yes (to some extent, very recent)

Syscall Example

- Using truss,

Only examine 1 process

```
$ truss date
execve("/usr/bin/date", 0x08047C9C, 0x08047CA4)  argc = 1
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
resolvepath("/usr/bin/date", "/usr/bin/date", 1023) = 13
xstat(2, "/usr/bin/date", 0x08047A58)           = 0
open("/var/ld/ld.config", O_RDONLY)              = 3
fxstat(2, 3, 0x08047988)                         = 0
mmap(0x00000000, 152, PROT_READ, MAP_SHARED, 3, 0) = 0xFEFB0000
close(3)                                         = 0
mmap(0x00000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, -1
sysconfig(_CONFIG_PAGESIZE)                      = 4096
[...]
```

Output is limited to provided options

truss *slows down* the target (probe effect)

Syscall Example

- Using DTrace

You can select which syscall(s)

You choose the output

```
# dtrace -n 'syscall:::entry { printf("%16s %x %x", execname, arg0, arg1); }'
dtrace: description 'syscall:::entry ' matched 233 probes
CPU     ID                    FUNCTION:NAME
  1  75943                       read:entry             Xorg f 8047130
  1  76211                  setitimer:entry             Xorg 0 8047610
  1  76143                     writev:entry             Xorg 22 80477f8
  1  76255                    pollsys:entry             Xorg 8046da0 1a
  1  75943                       read:entry             Xorg 22 85121b0
  1  76035                      ioctl:entry      soffice.bin 6 5301
  1  76035                      ioctl:entry      soffice.bin 6 5301
  1  76255                    pollsys:entry      soffice.bin 8047530 2
[...]
```

Minimum performance cost

Watch every process

DTrace Features*

- Dynamic Instrumentation
- Unified Instrumentation
- Arbitrary-context kernel instrumentation
- Data integrity
- Arbitrary actions
- Predicates
- High-level control language
- Scalable data aggregation
- Speculative tracing
- Scalable architecture
- Virtualized consumers

*http://www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf

What is DTrace For?

- Troubleshooting performance problems
    - Profile applications and the kernel
    - latency measurements
    - Looking for areas for improvement even when performance is acceptable
- Troubleshooting software bugs
    - Proving what the problem is, and isn't.
    - Measuring the magnitude of the problem.
- Detailed observability
    - Observing the kernel
    - Observing devices, such as disk or network activity.
    - Observing applications, whether they are from Sun, 3rd party, or in-house.

A Few Words on Operating System Support of DTrace...

DTrace in Solaris/OpenSolaris

- Check out Bryan's blog on DTrace's 5th birthday for some cool history and trivia

  http://blogs.sun.com/bmc/
- DTrace was integrated into Solaris 10, and available with Solaris 10 3/05
- Additional features added in subsequent releases
  - **OpenSolaris** on the leading edge
- Use the Wiki site for most recent documentation

  http://wikis.sun.com/display/DTrace/Documentation
- Solaris Process Privileges enable non-root users to use DTrace (e.g. in zones!)

  dtrace_user, dtrace_proc, dtrace_kernel

DTrace in Mac OS X

- Added to Leopard (10.5)
- Not all providers implemented
  - e.g. *sched* not there...
  - some are *intentionally omitted* (DRM issue!)
- **Instruments** is built on DTrace
- pid provider, and plockstat are implemented!

```
macosx> plockstat -A -p 37476
^C
Mutex hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
 1057    55473 0x16886fc4                   0x1fec60
   62   490985 0x605845c                    0x1fec60
  741    20183 0x58395cc                    0x1fec60
   72   194605 0x58395cc                    0x1fec60
    5   558552 0x605845c                    0x1fec60
   50    52090 0x16886fc4                   0x1fec60
```

DTrace in FreeBSD

- DTrace is available in FreeBSD, beginning with the 7.1 beta bits
  - Currently downloadable from the FreeBSD site
- After installing, you need to do a kernel build, reboot, and load the dtrace module

  http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/dtrace-enable.html
  - Don't forget “kldload dtraceall” after you reboot! (I did, even though it's in the documentation!)
- Several providers not yet implemented
- Thus far, I have limited experience with the FreeBSD DTrace functionality

```
freebsd# uname -a
FreeBSD freebsd.localdomain 7.1-BETA FreeBSD 7.1-BETA #0: <...> i386
freebsd# dtrace -l | wc -l
   33198

freebsd#
```

DTrace Components

- Probes
  - A point of instrumentation and data generation
- Providers
  - A major component of DTrace, Providers manage probes of specific types, and for a specific area of the system
    - syscall, io, sched, proc, vminfo, etc
- Consumers
  - Users of the framework

```
dtrace(1), lockstat(1), plockstat(1),
intrstat(1)
```

DTrace User Components

- Predicates
  - User-defined conditional statements evaluated when probes fire
  - Provides a control flow mechanism for your D programs – data pruning at the source
- Actions
  - What to do when the probe fires

Data to gather

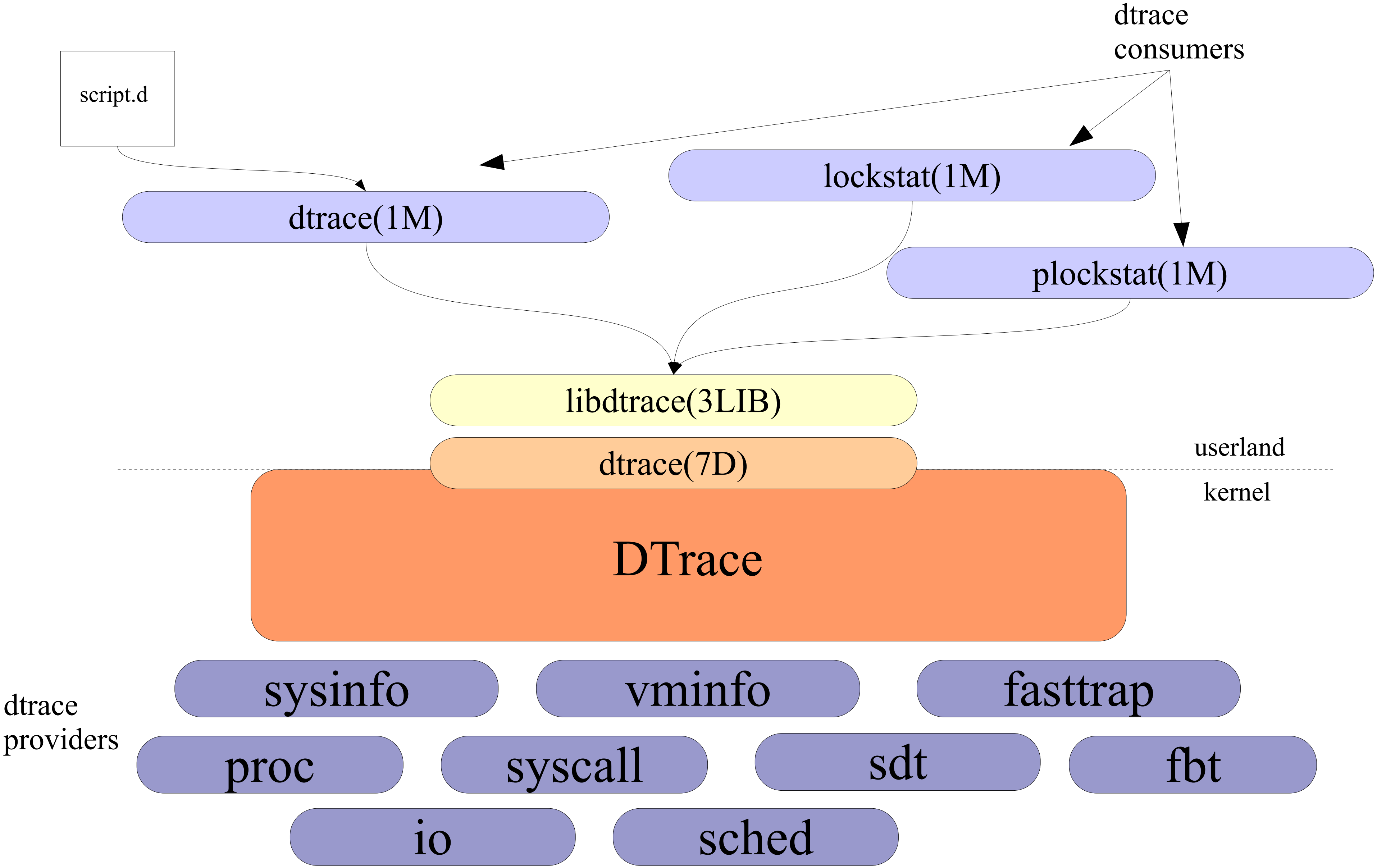
Timestamps for profiling

Many other actions supported

DTrace – What Happens

- *dtrace* command compiles the D language Script.
- Intermediate code checked for safety (like java).
- The compiled code is executed in the kernel by DTrace.
- DTrace instructs the provider to enable the probes
- As soon as the D program exits all instrumentation removed
- No limit (except system resources) on number of D scripts that can be run simultaneously
- Different users can debug the system simultaneously without causing data corruption or collision issues.

DTrace – The Big Picture



DTrace – On The Inside - Safety

- Inside interpreter: in the kernel space that interprets instructions and verifies that each pointer is safe to access or read
- **Protection against memory violations** – accessing a userland memory address results in a disabled probe
- **No loops**, avoids the *Halting Problem*
  - “Given a description of a program and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting. We say that the halting problem is undecidable over Turing machines.”
  - http://en.wikipedia.org/wiki/Halting_problem

DTrace Safety – A Bit More...

- *“...the most fundamental is the principle of* ***safety****: DTrace must not be able to accidentally induce system failure.”* *
- Probes are provided by instrumentation **providers** that guarantee their safety *
  - Users are not permitted to arbitrarily select instrumentation points
- While in probe context, DTrace itself must not call into any facilities in the kernel-at-large *
  - Probe context – protection against recursion
- Safe execution of user probe actions and predicates
  - Non-native execution: runs in a virtual machine
  - DTrace D programs compiled into a safe intermediate format for execution, and validated for safety

* http://blogs.sun.com/bmc/entry/dtrace_safety - Bryan Cantrill

Running DTrace

- **Only root** allowed to run DTrace by default
  - Solaris, OS X and FreeBSD
- In Solaris, process privileges can grant dtrace permission to non-root users;

```
$ ppriv -l | grep dtrace
```

`dtrace_kernel`	Allow DTrace kernel-level tracing
`dtrace_proc`	Allow DTrace process-level tracing. Allow process-level tracing probes to be placed and enabled in processes to which the user has perms
`dtrace_user`	Allow DTrace user-level tracing. Allow use of the syscall and profile DTrace providers to examine processes for which the user has permissions

  - Enable using usermod utility

```
# usermod -K defaultpriv=basic,dtrace_kernel,\
dtrace_proc,dtrace_user username
```

DTrace Framework

- Probes and Providers
- Actions and Predicates
- The D language
- Aggregations
- Pointers and Arrays
- Strings
- Structs and Unions
- Output formatting
- Speculative tracing

DTrace Probes

- A point of instrumentation, made available by a provider, which has a unique name
- A four-tuple name uniquely identifies every probe;

  provider:module:function:name

  - *provider*: the DTrace provider that manages the probe (DTrace kernel module)
  - *module*: kernel module or user library where the probe is located
  - *function*: kernel or user function containing the probe
  - *name*: represents an entry point in that function (e.g. *entry* or *return*), or has a meaningful name (e.g. io:::start, proc:::exec)
  - missing component means wildcard

Probes

- Anchored Probes
  - Instrument a specific point in code, e.g.

    `fbt:ufs:ufs_read:entry`

    `io:::start`

    `ip:::receive`
- Unanchored Probes
  - Are not associated with a specific location in code
  - Do not have a module or function component to their name
  - profile and tick

    `profile-997hz, tick-10sec`

Probes

- List probes
  - Use dtrace(1M) with the '-l' option
  - For each probe the four-tuple name will be displayed, probe components are ':' separated
  - List all probes:

```
$ dtrace -l
```

  - List all probes offered by syscall provider:

```
$ dtrace -lP syscall
```

  - List all probes offered by the ufs module:

```
$ dtrace -lm ufs
```

  - List all providers:

```
$ dtrace -l | awk '{print $2}' | sort -u
```

Probes

  - List all read function probes:

```
$ dtrace -l -f read
```

- Enabling probes
  - Activate a probe by not using '-l' option
  - Default action with enabled probes- the CPU, the probe number and name are displayed whenever the probe fires
  - Enable all probes from nfs and ufs module:

```
$ dtrace -m nfs,ufs
```

  - Enable all read function probes:

```
$ dtrace -f read
```

  - Enable all probes from io provider:

```
$ dtrace -P io
```

Probes

- BEGIN and END
  - BEGIN: fires each time a trace request is made

```
# dtrace -n BEGIN
dtrace: description 'BEGIN' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
^C
```

  - END: fires when the trace finishes

```
# dtrace -n END
dtrace: description 'END' matched 1 probe
^C
CPU     ID                    FUNCTION:NAME
  0      2                             :END
```

Probes

- ERROR
  - The ERROR probe fires when a runtime error is encountered

```
dhcp-s> dtrace -n 'dtrace:::BEGIN { myvar = *(char *)NULL; } dtrace:::ERROR { printf("OOoopppsss....\n"); }'
dtrace: description 'dtrace:::BEGIN ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1      3                           :ERROR OOoopppsss....

dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address (0x0) in action #1 at DIF offset 16
^C


dhcp-s> dtrace -qn 'dtrace:::BEGIN { myvar = *(char *)NULL; } dtrace:::ERROR { printf("OOoopppsss....\n"); }'
OOoopppsss....
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address (0x0) in action #1 at DIF offset 16

^C
```

Probes

- Examples
  - `syscall:::`
  - `syscall:::entry`
  - `syscall:::return`
  - `syscall::read:entry{ printf(“Process %d”, pid); }`
  - `syscall::write:entry/execname=="firefox-bin"/ { @[probefunc] = count(); }`
  - `sysinfo:::readch{ trace(execname); exit(0); }`
  - `sysinfo:::writech`
  - `io:::`

Probes and DTrace Built-in Variables

- Among the many built-in variables provided by DTrace, there are probe-specific variables available when a probe fires
  - Function call argument list. Arguments passed to a function instrumentable through an “entry” probe are available as either:

    int64_t arg0, arg1, ..., arg9 – args available as raw 64 bit integers

    args[0], args[1], ..., args[9] – typed args, corresponding to the specific data types of the arg list
- Probes in some providers have a specific arg list made available by the provider
  - e.g. the IO provider arg list of pointers to a buf structure, devinfo structure and fileinfo structure when IO provider probes fire
- **You need to RTFM to determine what args are available for a given provider !!!**
  - For function entry arg lists, you need man pages, kernel source, or just mdb(1) on a running Solaris system

Providers

- A methodology for instrumenting the system
- Providers offer all probes to the DTrace framework
- DTrace framework confirms to providers when a probe is activated
- Providers pass the control to DTrace when a probe is enabled
- Example of providers: syscall, lockstat, fbt, io, mib

Providers

- Providers do a couple interesting things for us...
  - Manage probes
  - Abstract a complex subsystem with intuitive probes, enabling and enhancing observability and analysis

    sched:::oncpu

    io:::start

    etc...
  - You can use DTrace effectively to track application and kernel activity in areas of the kernel that you may not be familiar with

Provider Documentation

- Some providers assume a little background knowledge, other providers assume a lot. Knowing where to find supporting documentation is important.
- Where do you find documentation on -
  - Syscalls?
  - User Libraries?
  - Application Code?
  - Kernel functions?

Provider Documentation

- Additional documentation may be found here,

Target	**Provider**	**Additional Docs**
syscalls	`syscall`	man(2)
libraries	`pid:lib*`	man(3C)
app code	`pid:a.out`	source code, ISV, developers
raw kernel	`fbt`	Solaris Internals 2nd Ed, http://cvs.opensolaris.org

Providers

```
nv98> dtrace -l | awk '{ print $2}' | sort -u
PROVIDER
Xserver767
dtrace
fbt
fsinfo
io
ip
lockstat
lx-syscall
mib
proc
profile
sched
sdt
syscall
sysevent
sysinfo
vminfo
```

```
macosx> dtrace -l | awk '{ print $2}' | sort -u
PROVIDER
dslockstat87530
dtrace
fbt
io
lockstat
mach_trap
mds66
plockstat16190
plockstat16191
plockstat16192
proc
profile
syscall
vminfo
macosx>
```

Providers – dtrace

- dtrace
  - Aforementioned BEGIN, END, ERROR probes
  - Useful for printing headers (BEGIN), data summary (END), and gathering more information on errors (ERROR)
  - The ERROR probe provides args with additional information
    - arg1 – EPID of probe that caused the error
    - arg2 – Index of the action that caused the fault
    - arg3 – DIF action
    - arg4 – Fault type
    - arg5 – Value particular to fault type

Providers - syscall

- Manages probes where “applications meet the kernel”
- Two probes for each system call
  - entry
  - return
- Arguments
  - entry – arg0...argn – the arg list to the system call
  - return – arg0 and arg1 – return value
  - D variable errno provide system call failure info
- Note some system calls do not directly map to syscall probefuncs
  - System V IPC

Providers – profile & tick

- Time-based interrupt firing
  - profile – fires on all CPUs
  - tick – fires on only 1 CPU
- Specify interval in probe
  - hz, sec or s, min or m, msec or ms, usec or u, etc
- Two args
  - arg0 – PC if in the kernel (sys mode)
  - arg1 – PC if in user (usr mode)
  - Very handy for system-wide profiling...

```
profile-997hz / arg0 != 0 / { … }    Am I in the kernel?
profile-997hz / arg1 != 0 / { … }    Am I in user land?
```

Provider - tick take 2

- tick-nnn also handy for
  - Building scripts that provide output at intervals (like the *stat commands)

```
tick-1sec { print(data); clear(data); }
```

  - Bail-out mechanism

```
tick-500ms { print(data); exit(0); }
```

Providers - sdt

- Statically Defined Tracing
  - Probes inserted at points of interest in the kernel
  - Allows the programmer to add probes to code with meaningful names without creating a new provider, using DTrace macros (sys/sdt.h);

```
DTRACE_PROBE(name);
DTRACE_PROBE1(name, type1, arg1);
DTRACE_PROBE2(name, type1, arg1, type2, arg2);
DTRACE_PROBE3(name, type1, arg1, type2, arg2, type3, arg3);
DTRACE_PROBE4(name, type1, arg1, type2, arg2, type3, arg3, type4, arg4);
```

-

Providers - fbt

- Function boundary tracing
  - Enable probes at kernel function entry and return points
  - Use requires some knowledge of the kernel
- Args
  - On entry probes, the arguments passed to the function are available as;

    args[] array – typed

    arg0 … argn – int64_t's
  - On return probes, function return values available in args[1]

Kernel Function Args... mdb(1) & dtrace(1) – Perfect Together

```
# mdb -k
Loading modules: [ unix krtld genunix specfs dtrace ufs sd ip sctp usba fcp fctl nca nfs random sppp lofs crypto
ptm logindmux md isp cpc fcip ipc ]
> ufs_read::nm -f ctype
C Type
int (*)(struct vnode *, struct uio *, int, struct cred *, struct caller_context *)
> ::print -t struct vnode
{
    kmutex_t v_lock {
        void * [1] _opaque
    }
    uint_t v_flag
    uint_t v_count
    void *v_data
    struct vfs *v_vfsp
    struct stdata *v_stream
    enum vtype v_type
    dev_t v_rdev
    struct vfs *v_vfsmountedhere
    struct vnodeops *v_op
    struct page *v_pages
    pgcnt_t v_npages
    ...
    char *v_path
    ...
}
```

```
# dtrace -n 'ufs_read:entry { printf("%s\n",stringof(args[0]->v_path));}'
dtrace: description 'ufs_read:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1  16777                   ufs_read:entry /usr/bin/cut

  1  16777                   ufs_read:entry /usr/bin/cut

  1  16777                   ufs_read:entry /usr/bin/cut

  1  16777                   ufs_read:entry /usr/bin/cut

  1  16777                   ufs_read:entry /lib/ld.so.1

  1  16777                   ufs_read:entry /lib/ld.so.1
....
```

Providers - sysinfo

- DTrace probes that enable gathering values of kernel statistics – sys kstats;

```
nv98> kstat -n sys
module: cpu                                 instance: 0
name:   sys                                 class:    misc
bawrite                             139
bread                               1122
bwrite                              1418
canch                               66
. . .
```

- Args

arg0 – Value by which the statistic will be incremented

arg1 – Pointer to the current value

arg2 – pointer to the cpu_t of the CPU the statistic is being incremented on

Providers - vminfo

- Similar to sysinfo – probes that correspond to named vm kstats
  - arg0 – value by which the stat will be incremented
  - arg1 – pointer to the current value of the stat
- Enables correlation of virtual memory events to processes/threads

```
#dtrace -n 'vminfo / execname != “dtrace” / { @vm[execname]=count(); }'
```

Providers - proc

- Events related to processes
  - create, exec, lwp-create, signals, etc
- The args vary, depending on which specific probe is enabled

/usr/demo/dtrace/whoexec.d

Providers - pid

- Using DTrace to look up into userland!
  - No code modifications required – it's all dynamic!

    pid1234:shared_object:function:name

    pid3402:libc:malloc:entry
- PIDs can be set using the DTrace $target macro
  - Set when either -c <command> or -p <PID> is used

    'pid$target:::entry { @[probemod, probefunc] = count() }' -c date

Providers - plockstat

- User level lock statistics
  - Similar to what lockstat(1) does for kernel lock stats
  - User mutex locks and Reader/Writer locks
- Check out the -V option...
  - Will generate the actual D executing...

```
plockstat -V -A -p 840 > pl.out 2>&1
```

Providers

- **io**
  - disk input and output requests
  - I/O by device, process, size, filename
- **mib**
  - counters for management information bases
  - IP, IPv6, ICMP, IPSec
- **sched**
  - kernel scheduler events
  - on-cpu, off-cpu, resume, preempt

Providers

- fsinfo
    - file system operations of interest
- ip
    - network events (packet send/receive)

Providers

- DTrace refers to most providers as “Stable” providers
  - The probes and args will not change across releases
  - Provides for building a toolbox that will work indefinitely
  - io, sched, proc, vminfo, sysinfo, fpuinfo, mib, etc, are all stable providers
  - fbt is not, since fbt by definition instruments the kernel functions entry and return points.
  - It is generally recommended to stick with stable providers, at least while you're getting started
  - Check the documentation for the specific stability level of a provider
  - New providers under development!

Providers, cont.

- Examples
  - **`proc:::exec`**
  - **`sched:::oncpu`**
  - **`fbt:ufs:ufs_read:entry`**
  - **`syscall`**`::read:entry{ printf(“Process %d”, pid); }`
  - **`syscall`**`::write:entry/execname=="firefox-bin"/ { @[probefunc] = count(); }`
  - **`sysinfo`**`:::readch{ trace(execname); exit(0); }`
  - **`sysinfo`**`:::writech`
  - **`io`**`:::start`

The D language

- A simple **(?)** dynamically interpreted language used by dtrace(1M)
- Similar to C language and awk(1):
  - Supports ANSI C operators and has support for strings
  - Supports several variable types, including built-in variables: pid, execname, timestamp, curthread, etc
- **No control-flow constructs**:
  - loops, if statements
- Arithmetic may only be performed on integers in D programs, floating-point arithmetic is not permitted in D

A DTrace D Program

```
probe
/ optional predicate /
{
    clause
    what to do when the probe(s) fire, and the predicate,
    if present, evaluates true
}
```

Example;

```
syscall::read:entry
/ execname == “java” /
{
    @reads[pid, fds[arg0].fi_pathname] = count();
}
```

Or, via the command line;

```
#dtrace -n 'syscall::read:entry / execname == “java” /
    { @reads[pid, fds[arg0].fi_pathname] = count(); }'
```

The D language, cont.

- Data Types
  - **Integer types**

    char
    short
    int
    long
    long long
  - **Float types**

    float
    double
    long double
  - **String type**

    string

The D language, cont.

- Operators
  - Arithmetic Operators, similar as in ANSI C

    `+ - * / %`

    may only be performed on integer operands, or on pointers

    not applicable on floats
  - Relational Operators

    `>, >=, <, <=, ==, !=`
  - Logical Operators

    `&&, ||, ^^`
  - Assignment Operators, similar as in ANSI C

    `=, +=`, ANSI-C compliant

The D language, cont.

- Variables: no need to declare them
- **Scalar Variables**
  - represents integers, strings, pointers
  - Three different types that define the variable scope;

    Global

    Thread-Local

    Clause-Local
  - created automatically – D figures out the type

The D Language

- Global variables
  - Visible in every clause of the D program
  - name and data storage location define once

```
dtrace:::BEGIN
{
    x = 123;
}

int n;
dtrace:::END
{
    n = 456;
    printf(“n: %d, x: %d\n”,n,x);
}
```

Global variable x

Explicit variable declaration, not needed. You can do this outside probe clause

The D language, cont.

- **Thread-local variables**
  - Variable storage local to each OS thread
  - Useful for setting trace flags
  - Use the “`self->`” identifier to declare a thread-local variable
  - Example which associates a thread-local variable called flag in function entry to trace desired kernel thread in corresponding return function

```
syscall::write:entry
/ pid == 3406 /
{
    self->flag = 1;
}
syscall::write:return
/ self->flag /
{
    self->flag = 0;
    ....
```

The D language, cont.

- Thread-local variables useful for computing the time spent in functions
- Example:

```
syscall::read:entry
{
    self->st = timestamp;
}
syscall::read:return
/ self->st /          /* this is the same as “self->st != 0” */
{
    self->rt = timestamp – self->st;
    self->st = 0;
    printf(“PID %d, read time: %d\n”, pid, self->rt);
}
```

The D language, cont.

- **Clause-Local Variables**
  - Their storage is reused for each program clause
  - Similar to automatic variables in a C, C++, or Java language
  - Are created on their first assignment
  - Referenced and assigned by using “`this->`” operator

```
BEGIN
{
 this->secs = timestamp / 1000000000;
 ...
}
```

The D language, cont.

- **Associative Arrays**
  - Collection of data elements
  - No predefined number of elements
  - Used to simulate hashes or data dictionaries
  - Very simple to use and different than a scalar array
  - Defined as: name[key] = expression

e.g.: `a[123,"abc"] = 456`

(a is associative array: a[int, string] stores an integer)

The D language, cont.

- **Built-in Variables**
  - pid: the current process ID
  - execname: the current executable name
  - timestamp: the time since boot, in nanoseconds
  - curthread: the current thread
  - probeprov, probemod, probefunc and probename identify the current probe name fields
- **External Variables**
  - used in some other parts: OS, kernel modules. e.g: \`kmem_flags, \`physmem

The D language, cont.

- Scripting in D
- Easy to create D scripts to hold one or more probe clauses
- All D scripts end in dot d (script_name.d)
- Add the interpreter as the first line in the script

```
#!/usr/sbin/dtrace -s
```

- Or create the script and run as;

```
#dtrace -s ./script.d
```

Actions & Subroutines

- Taken when a probe fires
- Indicated by following a probe specification with “*{ action }*”
- Actions trace data and modify state external to DTrace
  - Data recording actions operate on the principle buffer
  - The default action when a probe fires is to generate the CPU ID the probe fired on, the numeric ID of the DTrace probe, and the probe function and name
- Subroutines affect internal DTrace state

Actions, cont.

- Data Recording Actions
  - **trace(expression)**

    records the result of trace to the directed buffer

    `trace(pid)` traces the current process id

    `trace(execname)` traces the current application name
  - **printf()**

    traces a D expression

    allows output style formatting

    `printf(“execname is %s”, execname);`
  - **printa(aggregation)**

    used to display and format aggregations

    `printa(@agg1)`

    `printa(“%-@32s, %-@8d\n”,@execs, @pids);`

Actions, cont.

- Data Recording Actions
  - **stack()**

    records a kernel stack trace

    ```
    dtrace -n 'syscall::open:entry{ stack(); }'
    ```
  - **ustack()**

    records a user process stack trace

    allows to inspect userland stack processes

    ```
    dtrace -n 'syscall::open:entry{ ustack(); }' -c ls
    ```
  - **jstack()**

    similar with ustack(), but specifically for Java

    more space for deeper stack frames and longer symbol strings

Actions, cont.

- **Destructive Actions**
  - used to change the state of the system
  - use with caution, **it is disabled by default!!**

Process Destructive	**Results**
stop()	Stops the process which has executed the probe
raise()	Used to signal a process at a precise point during execution
copyout, copyoutstr()	
system()	
Kernel Destructive	**Results**
breakpoint()	Stops the system abd transfers the control to the kernel debugger
panic()	Triggers a panic. Used to force a crash dump
chill()	A sophisticated routine to inject a short delay. Used for timings measurements

Actions, cont.

- **Special Actions**
  - exit(int) - stop tracing and exits

  - Other subroutines:

    alloca() – allocates a n size bytes buffer
    basename() - formats the path names
    copyin() - creates a buffer and returns its address
    copyinstr() - creates a buffer and returns its address
    rand() - returns a weak pseudo-random number
    strlen() - returns the length of a string in bytes
    strjoin() - returns a string as a concatenation of str1 and str2

Actions, cont.

- Examples
  - `syscall:::`
  - `syscall:::entry`
  - `syscall:::return`
  - `syscall::read:entry{ printf(“Process %d”, pid); }`
  - `syscall::write:entry/execname=="firefox-bin"/ { @[probefunc] = count(); }`
  - `sysinfo:::readch{ trace(execname); exit(0); }`
  - `sysinfo:::writech`
  - `io:::`

Predicates

- D expressions that define a conditional test
- Allow actions to only be taken when certain conditions are met. A predicate has this form: */predicate/*
- The actions will be activated only if the value of the predicate expression is true
- Used to filter and meet certain conditions: look only for a process which has the pid = 1203, match a process which has the name firefox-bin

Predicates, cont.

- Examples
  - syscall:::
  - syscall:::entry
  - syscall:::return
  - syscall::read:entry{ printf(“Process %d”, pid); }
  - syscall::write:entry**/execname=="firefox-bin"/** { @[probefunc] = count(); }
  - sysinfo:::readch{ trace(execname); exit(0); }
  - sysinfo:::writech
  - io:::

Aggregations

- Used to aggregate data and look for trends
- Simple to generate reports about: total system calls used by a process or an application, the total number of read or writes by process...
- Has the general form:

@name[keys] = aggfunc(args)

- There is no need to use other tools like: awk(1), perl(1)
- The general definition of aggregating function:

$$f(f(x_0) \cup f(x_1) \cup \ldots \cup f(x_n)) = f(x_0 \cup x_1 \cup \ldots \cup x_n)$$

Aggregations

- Aggregating functions
  - count() : the number of times called, used to count for instance the total number of reads or system calls
  - sum() : the total value of the specified expressions
  - avg() : the arithmetic average of the specified expression
  - min() : the smallest value of the specified expression
  - max(): the largest value of the specified expression
  - quantize() : a power-of-two frequency distribution, simple to use to draw distributions
- Non-aggregating functions
  - mode and median

Aggregations, cont.

- What's going on with my system ?

```
dtrace -n syscall:::entry
```

- Difficult to read, start aggregating...

```
dtrace -n 'syscall:::entry{@[execname] = count();}'
```

- Filter on read system call

```
dtrace -n
  'syscall::read*:entry{@[execname]=count();}'
```

- Add the file descriptor information

```
dtrace -n
  'syscall::read*:entry{@[execname,arg0]=count();}'
```

Aggregations, cont.

- Drill-down and get a distribution of each read by application name

```
syscall::read*:entry
{
 self ->ts=timestamp;
}

syscall::read*:return
/self -> ts/
{
 @time[execname] = quantize(timestamp - self->ts);
 self->ts = 0;
}
```

Aggregations, cont.

- Data normalization
  - used to aggregate over a specific constant reference: e.g.: system calls per second
  - normalize()
  - denormalize()
- Truncate
  - used to minimize the aggregation results, keep certain top results
  - trunc(aggregation, trunc value)

Pointers and Arrays

- Pointers determines which location in memory we are referencing
- Similar mechanism as in ANSI-C
- Safe access and control of pointers by DTrace
- Invalid memory access and alignment checks

```
BEGIN
{
 x = (int *)NULL;
 y=*x;
 trace(y);
}
```

Pointers and Arrays, cont.

- Support for scalar arrays, similar with C/C++
- Indexed from 0, fixed length
- Sometimes used to access certain OS array data structures
- Defined as: int a[int]

  Example: int a[4]; 4 elements: a[0], a[1], a[2], a[3]
- Scalar and associative arrays

Item	**Predefined Size**	**Consecutive storage order**	**Form**
Scalar Array	Yes	Yes	int a[4]
Associative Array	No	No	a[123,"abc"]

DTrace Framework

- Introduction
- Probes, Providers, Actions, Predicates
- The D language
- Aggregations
- Pointers and Arrays
- **Strings**
- Structs and Unions
- Output formatting
- Speculative tracing

Strings

- Support for strings in D
- Built-in data type very easy to use
- Strings constants defined between “ “
- String assignment using = operator
  - Example: *s = “my string”;*
- String comparation using the relational operators (<, >, <=, >=, ==, !=)
  - Example: execname == “firefox-bin”
- Comparation is done byte-by-byte as in C like in strcmp(3C) routine

DTrace Framework

- Introduction
- Probes, Providers, Actions, Predicates
- The D language
- Aggregations
- Pointers and Arrays
- Strings
- **Structs and Unions**
- Output formatting
- Speculative tracing

Output formatting

- Special routines to format the output: trace(), printf() or printa()
- For specific output format use built-in printf()
  - `printf(“execname is %s”, execname);`
  - `printf(“%d spent %d secs in read\n”, pid, timestamp - t);`
- For aggregations use printa()
  - `printa(“Aggregation is:”, @a);`
  - `printa(@count);`
- Basic trace()
  - `trace(execname);`

DTrace Methods and Use

- Learning the mechanics of DTrace is great, but DTrace is, after all, a tool
- Like any tool, it's usefulness depends on the skill set and experience of the user
- The great news is DTrace is really easy to use!
  - It's easy to so simple things in DTrace that tell you a LOT about what your system and application is doing
- With time and experience, you'll only get better at root-causing sticky problems

DTrace One Liners

- System Calls Count by Application

```
$ dtrace -n 'syscall:::entry{@[execname] =
  count();}'
```

- System Calls Count by Application and Process

```
$ dtrace -n 'syscall:::entry{@[execname,pid]
  = count();}'
```

- How many times a file has been opened

```
$ dtrace -n
  'syscall::open:entry{@[copyinstr(arg0)] =
  count();}'
```

DTrace One Liners

- Files Opened by process

```
$ dtrace -qn
  'syscall::open*:entry{ printf("%s
  %s\n",execname,copyinstr(arg0)); }'
```

- Read Bytes by process

```
$ dtrace -n 'sysinfo:::readch{ @[execname] =
  sum(arg0);}'
```

- Write Bytes by process

```
$ dtrace -n 'sysinfo:::writech{ @[execname]
  = sum(arg0);}'
```

DTrace One Liners, cont.

- How big a read is

```
$ dtrace -n 'syscall::read:entry{@[execname]
  = quantize(arg2);}'
```

- How big a write is

```
$ dtrace -n
  'syscall::write:entry{@[execname] =
  quantize(arg2);}'
```

- Disk size by process

```
$ dtrace -qn 'io:::start{printf("%d %s
  %d\n",pid,execname,args[0]->b_bcount); }'
```

DTrace One Liners, cont.

- High system time

```
$ dtrace -n profile-501'{@[stack()] =
  count()}END{trunc(@, 25)}'
```

- What processes are using fork

```
$ dtrace -n 'syscall::fork*:entry{printf("%s
  %d",execname,pid);}'
```

The DTraceToolkit

Brendan Gregg developed the toolkit
Stefan Parvu wrote the slides

DTraceToolkit

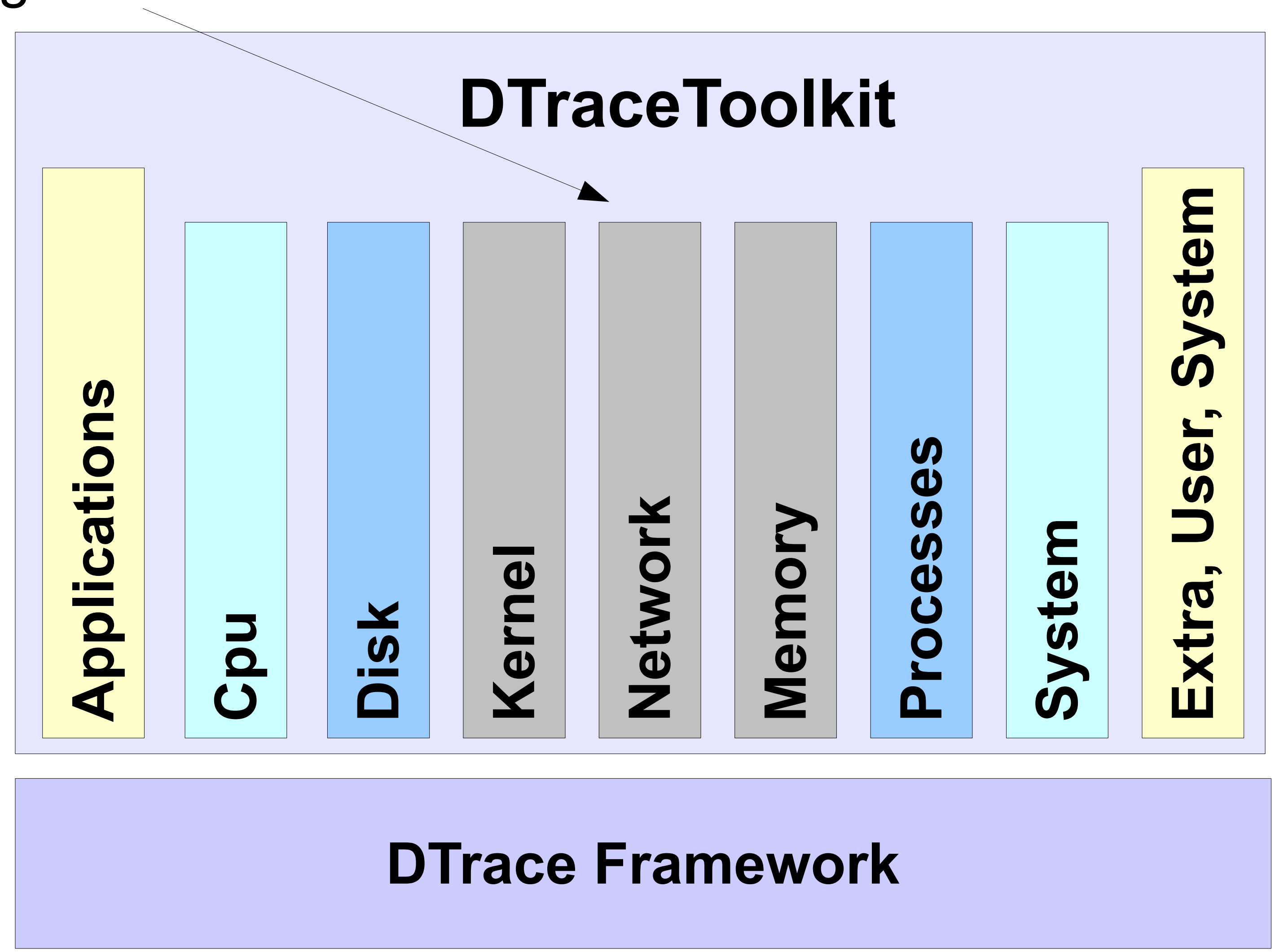
- **Introduction**
- Installation and Setup
- Toolkit elements
- Categories
- Free your mind
- Examples

Introduction

- The DTraceToolkit is a collection of useful documented scripts developed by the OpenSolaris DTrace community built on top of DTrace framework
- Available under www.opensolaris.org
- Ready DTrace scripts
- The toolkit contains:
  - the scripts
  - the man pages
  - the example documentation
  - the notes files
  - the tutorials

Introduction, cont.

Script Categories: collection of D scripts



DTraceToolkit

- Applications
- Cpu
- Disk
- Kernel
- Network
- Memory
- Processes
- System
- Extra, User, System

DTrace Framework

DTraceToolkit

- Introduction
- **Installation and Setup**
- Toolkit elements
- Categories
- Free your mind
- Examples

Installation and Setup

- Download the toolkit

  http://www.opensolaris.org/os/community/dtrace/dtracetoolkit

- Installation Notes
  - gunzip and "tar xvf" the file
  - run ./install – default installation /opt/DTT
  - read Guide to find out how to get started
  - a list of scripts is in Docs/Contents
- Setup DTT
  - PATH=$PATH:/opt/DTT/Bin
  - MANPATH=$MANPATH:/opt/DTT/Man

    (assuming the toolkit was installed in /opt/DTT)

DTraceToolkit

- Introduction
- Installation and Setup
- **Toolkit Elements**
- Categories
- Free your mind
- Examples

Toolkit Elements

```
DTraceToolkit-X.XX/
     Bin/                    Symlinks to the scripts
     Apps/                   Application specific scripts
     Cpu/                    Scripts for CPU analysis
     Disk/                   Scripts for disk I/O analysis
     Docs/                   Documentation
        Contents             Command list for the Toolkit
        Examples/            Examples of command usage
        Faq                  Frequently asked questions
        Links                Further DTrace links
        Notes/               Notes on Toolkit commands
        Readme               Readme for using the docs
     Extra/                  Misc scripts
     Guide                   This file!
     Kernel/                 Scripts for kernel analysis
     License                 The CDDL license
     Locks/                  Scripts for lock analysis
     Man/                    Man pages
        man1m/               Man pages for the Toolkit commands
     Mem/                    Scripts for memory analysis
     Net/                    Scripts for network analysis
     Proc/                   Scripts for process analysis
     System/                 Scripts for system analysis
     User/                   Scripts for user based activity analysis
     Zones/                  Scripts for analysis by zone
     Version                 DTraceToolkit version
     install                 Install script, use for installs only
```

Toolkit Elements, cont.

- Categories
  - Apps – scripts for certain applications: Apache, NFS
  - Cpu – scripts for measuring CPU activity
  - Disk – scripts to analyse I/O activity
  - Extra – other categories
  - Kernel – scripts to monitor kernel activity
  - Locks – scripts to analyse locks
  - Mem – scripts to analyse memory and virtual memory
  - Net – scripts to analyse activity of the network interfaces, and the TCP/IP stack
  - Proc – scripts to analyse activity of a process
  - System – scripts to measure system wide activity
  - User – scripts to monitor activity by UID
  - Zones – scripts to monitor activity by zone

Toolkit Elements, cont.

- Documentation
  - Man/: all scripts are documented as UNIX manual pages
  - Docs/: a generic place to find the documentation
  - Docs/Notes/: several short guides about toolkit's commands
  - Docs/Example/: examples of command usage
  - Docs/Content/: complete list of all commands
  - Docs/Faq/: DTT Frequently Asked Questions

DTraceToolkit

- Introduction
- Installation and Setup
- Toolkit Elements
- **Categories**
- Free your mind
- Examples

Categories

- **Applications**
  - Used to measure and report certain metrics from applications like: Apache Web server, NFS client, UNIX shell
  - **httpdstat.d**: computes real-time Apache web statistics: the number of connections, GET, POST, HEAD and TRACE requests
  - **nfswizard.d**: used to measure the NFS client activity regarding response time and file accesses
  - **shellsnoop**: captures keystrokes, used to debug and catch command output. Use with caution !
  - **weblatency.d**: counts connection speed delays, DNS lookups, proxy delays, and web server response time. Uses by default Mozilla browser

Categories, cont.

- **Cpu**
  - Reports and list the CPU activity like: cross calls, interrupt activity by device, time spent servicing interrupts, CPU saturation
  - **cputypes.d**: lists the information about CPUs: the number of physical install CPUs, clock
  - **loads.d**: prints the load average, similar to uptime
  - **intbycpu.d**: prints the number of interrupts by CPU
  - **intoncpu.d**: lists the interrupt activity by device; example: the time consumed by the ethernet driver, or the audio device
  - **inttimes.d**: reports the time spent servicing the interrupt

Categories, cont.

- **Cpu**
  - **xcallsbypid.d** – list the inter-processor cross-calls by process id. The inter-process cross calls is an indicator how much work a CPU sends to another CPU
  - **dispqlen.d** – dispatcher queue length by CPU, measures the CPU saturation
  - **cpuwalk.d** – identify if a process is running on multiple CPUs concurrently or not
  - **runocc.d** – prints the dispatcher run queue, a good way to measure CPU saturation

Categories, cont.

- **Disk**
  - Analyses I/O activity using the io provider from DTrace: disk I/O patterns, disk I/O activity by process, the seek size of an I/O operation
  - **iotop**: a top like utility which lists disk I/O events by processes
  - **iosnoop**: a disk I/O trace event application. The utility will report UID, PID, filename regarding for a I/O operation
  - **bitesize.d**: analyse disk I/O size by process
  - **seeksize.d**: analyses the disk I/O seek size by identifying what sort I/O operation the process is making: sequential or random

Categories, cont.

- **Disk**
  - **iofile.d**: prints the total I/O wait times. Used to debug applications which are waiting for a disk file or resource
  - **iopattern**: computes the percentage of events that were of a random or sequential nature. Used easily to identify the type of an I/O operation and the average, totals numbers
  - **iopending**: prints a plot for the number of pending disk I/O events. This utility tries to identify the "serialness" or "parallelness" of the disk behavior
  - **diskhits**: prints the load average, similar to uptime
  - **iofileb.d**: prints a summary of requested disk activity by pathname, providing totals of the I/O events in bytes

Categories, cont.

- **FS**
  - Analyses the activity on the file system level: write cache miss, read file I/O statistics, system calls read/write
  - **vopstat**: traces the vnode activity
  - **rfsio.d**: provides statistics on the number of reads: the bytes read from file systems (logical reads) and the number of bytes read from physical disk
  - **fspaging.d**: used to examine the behavior of each I/O layer, from the syscall interface to what the disk is doing
  - **rfileio.d**: similar with rfsio.d but reports by file

Categories, cont.

- **Kernel**
  - Analyses kernel activity: DNLC statistics, CPU time consumed by kernel, the threads scheduling class and priority
  - **dnlcstat**: inspector of the Directory Name Lookup Cache (DNLC)
  - **cputimes**: print CPU time consumed by the kernel, processes or idle
  - **cpudist**: print CPU time distributions by kernel, processes or idle
  - **cswstat.d**: prints the context switch count and average
  - **modcalls.d**: an aggregation for kernel function calls by module

Categories, cont.

- **Kernel**
  - **dnlcps.d**: prints DNLC statistics by process
  - **dnlcsnoop.d**: snoops DNLC activity
  - **kstat_types.d**: traces kstat reads
  - **pridist.d**: outputs the process priority distribution. Plots which process is on the CPUs, and under what priority it is
  - **priclass.d**: outputs the priority distribution by scheduling class. Plots a distribution
  - **whatexec.d**: determines the types of files which are executed by inspected the first four bytes of the executed file

Categories, cont.

- **Locks**
  - Analyses lock activity using lockstat provider
  - **lockbydist.d**: lock distribution by process name
  - **lockbyproc.d**: lock time by process name

Categories, cont.

- **Memory**
  - This category analyses memory and virtual memory things: virtual memory statistics, page management, minor faults
  - **vmstat.d**: a vmstat like utility written in D
  - **vmstat-p.d**: a vmstat like utility written in D which does display what “vmstat -p” does: reporting the paging information
  - **xvmstat**: a much improved version of vmstat which does count the following numbers: free RAM, virtual memory free, major faults, minor faults, scan rate

Categories, cont.

- **Memory**
  - **swapinfo.d**: prints virtual memory info, listing all memory consumers related with virtual memory including the swap physical devices
  - **pgpginbypid.d**: prints information about pages paged in by process id
  - **minfbypid.d**: detects the biggest memory consumer using minor faults, an indication of memory consumption

Categories, cont.

- **Network**
  - These scripts analyse the activity of the network interfaces and the TCP/IP stack. Some scripts are using the mib provider. Used to monitor incoming
  - **icmpstat.d**: reports ICMP statistics per second, based on **mib**
  - **tcpstat.d**: prints TCP statistics every second, retrieved from the **mib** provider: TCP bytes received and sent, TCP bytes retransmitted
  - **udpstat.d**: prints UDP statistics every second, retrieved from the **mib** provider
  - **tcpsnoop.d**: analyses TCP network packets and prints the responsible PID and UID. Useful to detect which processes are causing TCP traffic

Categories, cont.

- **Network**
  - connections: prints the inbound TCP connections. This displays the PID and command name of the processes accepting connections
  - **tcptop**: display top TCP network packets by process. It can help identify which processes are causing TCP traffic
  - **tcpwdist.d**: measures the size of writes from applications to the TCP level. It can help identify which process is creating network traffic

Categories, cont.

- **Process**
  - Analyses process activity: system calls/process, bytes written or read by process, files opened by process,
  - **sampleproc**: inspect how much CPU the application is using
  - **threaded.d**: see how well a multithreaded application uses its threads
  - **writebytes.d**: how many bytes are written by process
  - **readbytes.d**: how many bytes are read by process
  - **kill.d**: a kill inspector. What how signals are send to what applications
  - **newproc.d**: snoop new processes as they are executed

Categories, cont.

- **Process**
  - **syscallbyproc.d** & **syscallbypid.d**: system calls by process or by PID
  - **filebyproc.d**: files opened by process
  - **fddist**: a file descriptor reporter, used to print distributions for read and write events by file descriptor, by process. Used to determine which file descriptor a process is doing the most I/O with
  - **pathopens.d**: prints a count of the number of times files have been successfully opened
  - **rwbypid.d**: reports the no. of read/writes calls by PID
  - **rwbytype.d**: identifies the vnode type of read/write activity - whether that is for regular files, sockets, character special devices

Categories, cont.

- **Process**
  - **sigdist.d**: prints the number of signals received by process and the signal number
  - **topsysproc**: a report utility listing top number of system calls by process
  - **pfilestat**: prints I/O statistics for each file descriptor within a process. Very useful for debug certain processes
  - **stacksize.d**: measures the stack size for running threads
  - **crash.d**: reports about crashed applications. Useful to identify the last seconds of a crashed application
  - **shortlived.d**: snoops the short life activity of some processes

Categories, cont.

- **System**
  - Used to measure system wide activity
  - **uname-a.d**: simulates 'uname -a' in D
  - **syscallbysysc.d**: reports a total on the number od system calls on the system
  - **sar-c.d**: reports system calls usage similar to 'sar -c'
  - **topsyscall**: prints a report of the top system calls on the system

DTraceToolkit

- Introduction
- Installation and Setup
- Toolkit Elements
- Categories
- **Free your mind**
- Real Examples

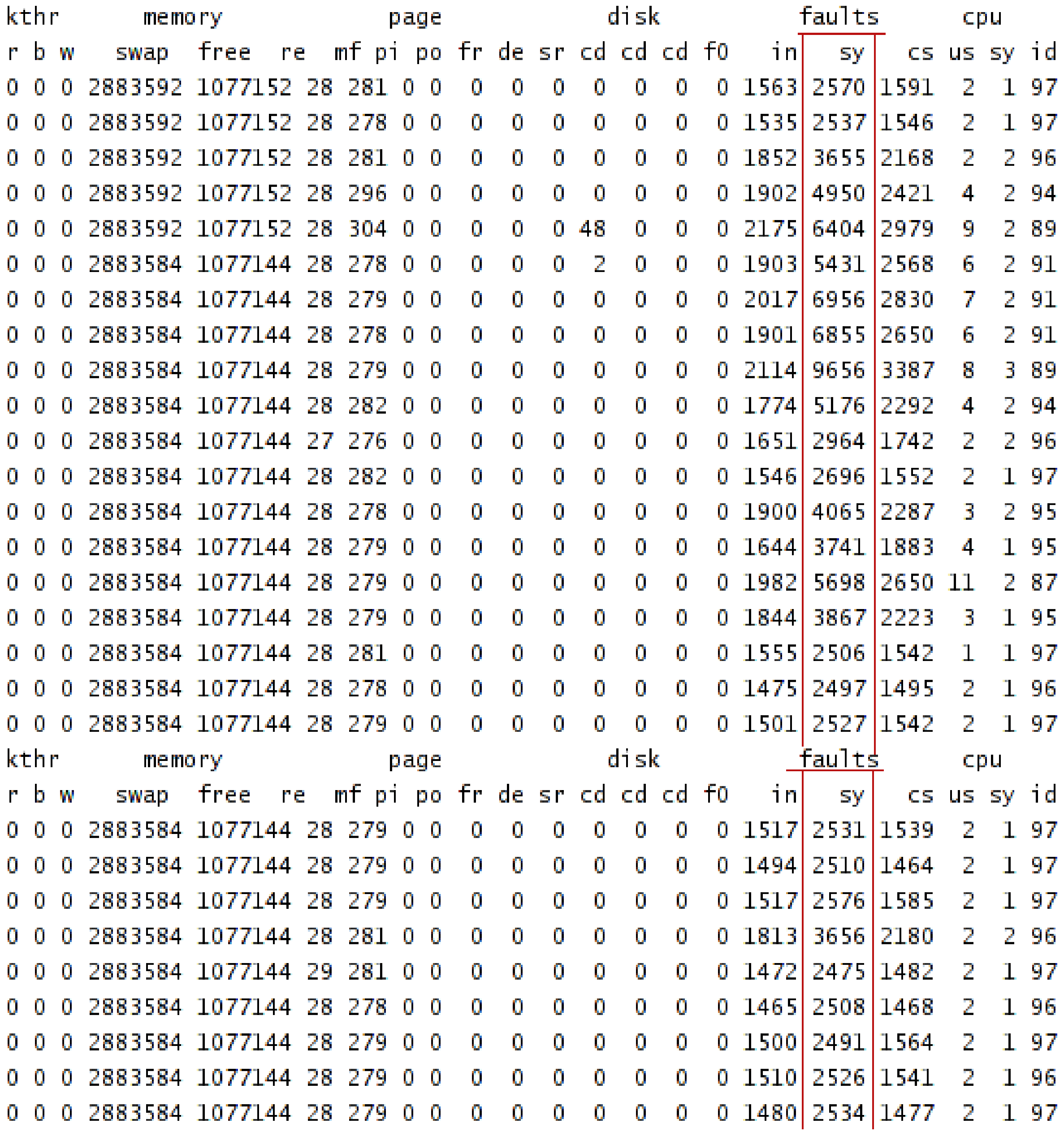
DTraceToolkit

- Introduction
- Installation and Setup
- Toolkit Elements
- Categories
- Free your mind
- **Real Examples**

1.High System Calls

- A case where *vmstat 1* reports a high number of system calls
- What to do ?
- Count the total number of system calls
- Use a simple DTrace aggregation to find out what application are responsible for that
- Think to enhance the aggregation for a better reporting or better...
- Use DTT utilities to find out what is going on, getting as well a nice report

1.High System Calls, cont.



```
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd cd cd f0   in   sy   cs us sy id
 0 0 0 2883592 1077152 28 281 0 0  0  0  0  0  0  0  0 1563 2570 1591  2  1 97
 0 0 0 2883592 1077152 28 278 0 0  0  0  0  0  0  0  0 1535 2537 1546  2  1 97
 0 0 0 2883592 1077152 28 281 0 0  0  0  0  0  0  0  0 1852 3655 2168  2  2 96
 0 0 0 2883592 1077152 28 296 0 0  0  0  0  0  0  0  0 1902 4950 2421  4  2 94
 0 0 0 2883592 1077152 28 304 0 0  0  0  0 48  0  0  0 2175 6404 2979  9  2 89
 0 0 0 2883584 1077144 28 278 0 0  0  0  0  2  0  0  0 1903 5431 2568  6  2 91
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 2017 6956 2830  7  2 91
 0 0 0 2883584 1077144 28 278 0 0  0  0  0  0  0  0  0 1901 6855 2650  6  2 91
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 2114 9656 3387  8  3 89
 0 0 0 2883584 1077144 28 282 0 0  0  0  0  0  0  0  0 1774 5176 2292  4  2 94
 0 0 0 2883584 1077144 27 276 0 0  0  0  0  0  0  0  0 1651 2964 1742  2  2 96
 0 0 0 2883584 1077144 28 282 0 0  0  0  0  0  0  0  0 1546 2696 1552  2  1 97
 0 0 0 2883584 1077144 28 278 0 0  0  0  0  0  0  0  0 1900 4065 2287  3  2 95
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1644 3741 1883  4  1 95
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1982 5698 2650 11  2 87
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1844 3867 2223  3  1 95
 0 0 0 2883584 1077144 28 281 0 0  0  0  0  0  0  0  0 1555 2506 1542  1  1 97
 0 0 0 2883584 1077144 28 278 0 0  0  0  0  0  0  0  0 1475 2497 1495  2  1 96
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1501 2527 1542  2  1 97
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd cd cd f0   in   sy   cs us sy id
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1517 2531 1539  2  1 97
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1494 2510 1464  2  1 97
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1517 2576 1585  2  1 97
 0 0 0 2883584 1077144 28 281 0 0  0  0  0  0  0  0  0 1813 3656 2180  2  2 96
 0 0 0 2883584 1077144 29 281 0 0  0  0  0  0  0  0  0 1472 2475 1482  2  1 97
 0 0 0 2883584 1077144 28 278 0 0  0  0  0  0  0  0  0 1465 2508 1468  2  1 96
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1500 2491 1564  2  1 97
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1510 2526 1541  2  1 96
 0 0 0 2883584 1077144 28 279 0 0  0  0  0  0  0  0  0 1480 2534 1477  2  1 97
```

1.High System Calls, cont.

- Start a simple aggregation:

```
$ dtrace -n 'syscall::entry{@[execname] =
 count();}'
```

- Select the top consumer and start aggregating again:

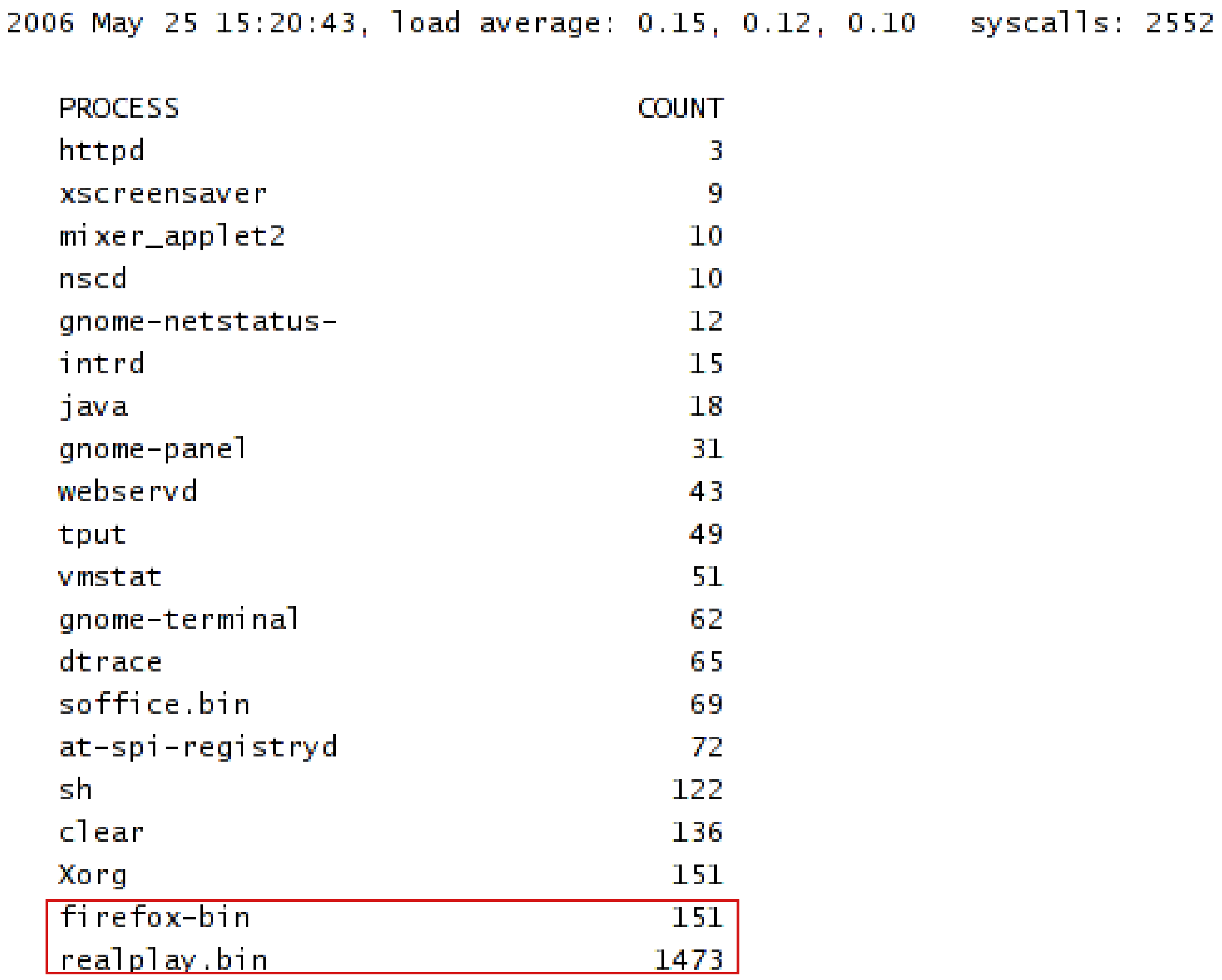
```
$ dtrace -n
 'syscall::entry/execname==”your-app”/
 {@[probefunc] = count();}'
```

- Count the number of system calls globally:

```
$ dtrace -n 'syscall::entry{@[probefunc]
 = count();}'
```

- Better run *topsysproc* from Proc Category

1.High System Calls, cont.



```
2006 May 25 15:20:43, load average: 0.15, 0.12, 0.10   syscalls: 2552

   PROCESS                                   COUNT
   httpd                                         3
   xscreensaver                                  9
   mixer_applet2                                10
   nscd                                         10
   gnome-netstatus-                             12
   intrd                                        15
   java                                         18
   gnome-panel                                  31
   webservd                                     43
   tput                                         49
   vmstat                                       51
   gnome-terminal                               62
   dtrace                                       65
   soffice.bin                                  69
   at-spi-registryd                             72
   sh                                          122
   clear                                       136
   Xorg                                        151
   firefox-bin                                 151
   realplay.bin                               1473
```

1.High System Calls, cont.

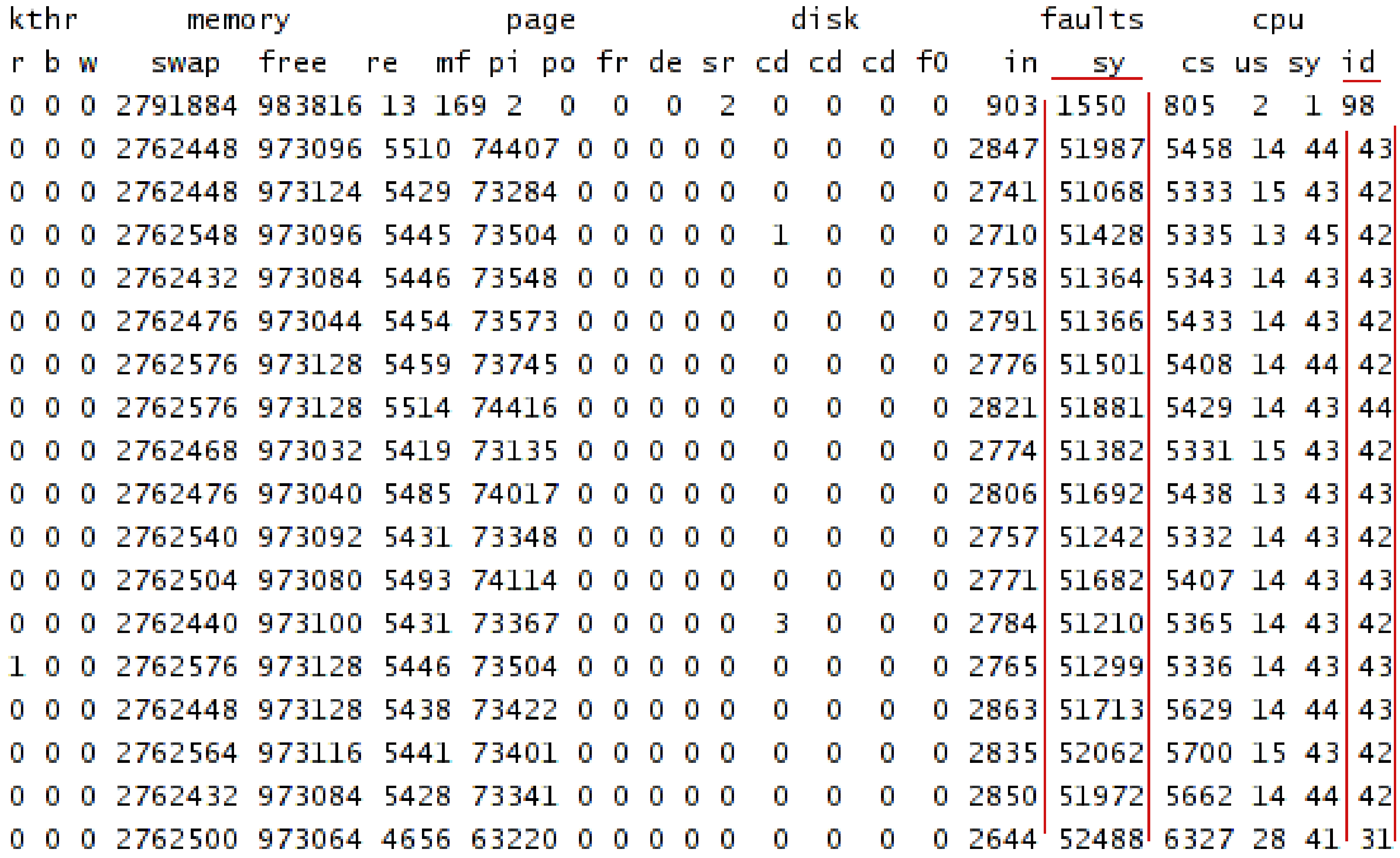
- **Conclusions:**
  - Not able to see who does all those system calls using basic utilities: *vmstat*, *iostat*, *prstat*
  - Easy to detect and get the report about the top system calls consumers using DTT utility: *topsysproc*

2.High CPU Utilization

- There is a high CPU utilisation under the system without any sign who is generating that
- What to do ?
- Does it help to run: *prstat*, *mpstat*, *vmstat*, *iostat* ?
- Solve the problem by using: *topsysproc*, and *execsnoop* from DTT

2.High CPU Utilisation, cont.

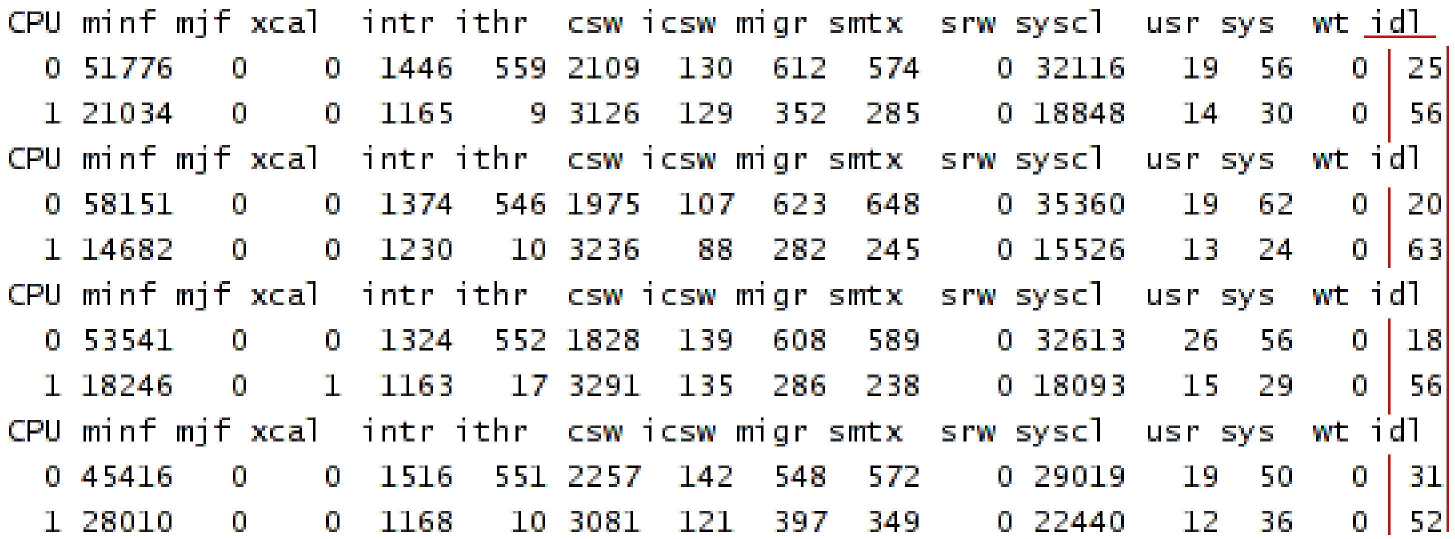
- The output from *vmstat 1*:



```
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd cd cd f0   in   sy   cs us sy id
 0 0 0 2791884 983816 13 169 2  0  0  0  2  0  0  0  0  903 1550  805  2  1 98
 0 0 0 2762448 973096 5510 74407 0 0 0 0 0  0  0  0  0 2847 51987 5458 14 44 43
 0 0 0 2762448 973124 5429 73284 0 0 0 0 0  0  0  0  0 2741 51068 5333 15 43 42
 0 0 0 2762548 973096 5445 73504 0 0 0 0 0  1  0  0  0 2710 51428 5335 13 45 42
 0 0 0 2762432 973084 5446 73548 0 0 0 0 0  0  0  0  0 2758 51364 5343 14 43 43
 0 0 0 2762476 973044 5454 73573 0 0 0 0 0  0  0  0  0 2791 51366 5433 14 43 42
 0 0 0 2762576 973128 5459 73745 0 0 0 0 0  0  0  0  0 2776 51501 5408 14 44 42
 0 0 0 2762576 973128 5514 74416 0 0 0 0 0  0  0  0  0 2821 51881 5429 14 43 44
 0 0 0 2762468 973032 5419 73135 0 0 0 0 0  0  0  0  0 2774 51382 5331 15 43 42
 0 0 0 2762476 973040 5485 74017 0 0 0 0 0  0  0  0  0 2806 51692 5438 13 43 43
 0 0 0 2762540 973092 5431 73348 0 0 0 0 0  0  0  0  0 2757 51242 5332 14 43 42
 0 0 0 2762504 973080 5493 74114 0 0 0 0 0  0  0  0  0 2771 51682 5407 14 43 43
 0 0 0 2762440 973100 5431 73367 0 0 0 0 0  3  0  0  0 2784 51210 5365 14 43 42
 1 0 0 2762576 973128 5446 73504 0 0 0 0 0  0  0  0  0 2765 51299 5336 14 43 43
 0 0 0 2762448 973128 5438 73422 0 0 0 0 0  0  0  0  0 2863 51713 5629 14 44 43
 0 0 0 2762564 973116 5441 73401 0 0 0 0 0  0  0  0  0 2835 52062 5700 15 43 42
 0 0 0 2762432 973084 5428 73341 0 0 0 0 0  0  0  0  0 2850 51972 5662 14 44 42
 0 0 0 2762500 973064 4656 63220 0 0 0 0 0  0  0  0  0 2644 52488 6327 28 41 31
```

2.High CPU Utilisation, cont.

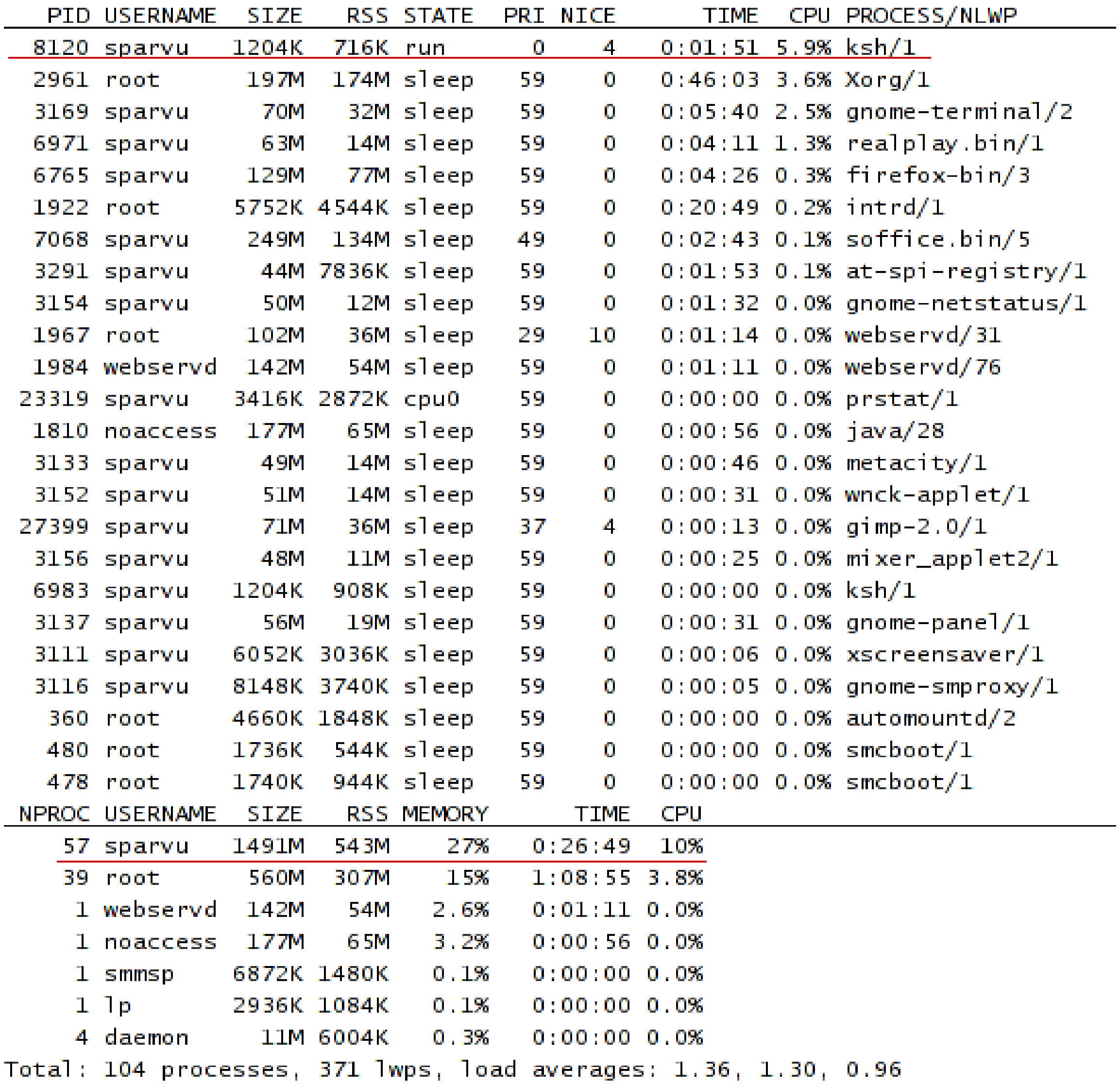
- The output from *mpstat 1*:



```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0 51776   0    0  1446  559 2109  130  612  574    0 32116   19  56   0  25
  1 21034   0    0  1165    9 3126  129  352  285    0 18848   14  30   0  56
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0 58151   0    0  1374  546 1975  107  623  648    0 35360   19  62   0  20
  1 14682   0    0  1230   10 3236   88  282  245    0 15526   13  24   0  63
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0 53541   0    0  1324  552 1828  139  608  589    0 32613   26  56   0  18
  1 18246   0    1  1163   17 3291  135  286  238    0 18093   15  29   0  56
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0 45416   0    0  1516  551 2257  142  548  572    0 29019   19  50   0  31
  1 28010   0    0  1168   10 3081  121  397  349    0 22440   12  36   0  52
```

2.High CPU Utilisation, cont.

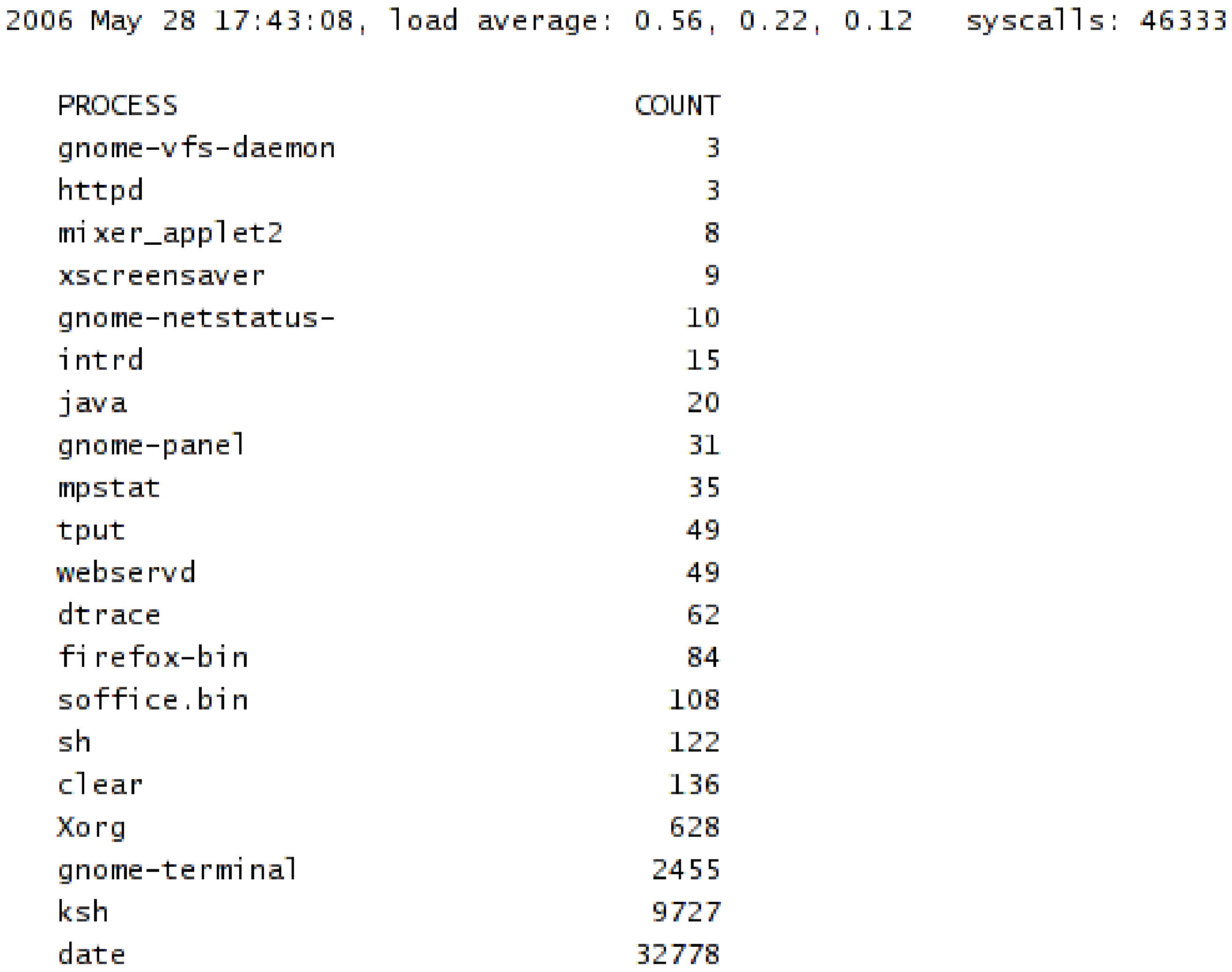
- The output from *prstat -a*:



```
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
  8120 sparvu   1204K  716K run      0    4   0:01:51 5.9% ksh/1
  2961 root      197M  174M sleep   59    0   0:46:03 3.6% Xorg/1
  3169 sparvu     70M   32M sleep   59    0   0:05:40 2.5% gnome-terminal/2
  6971 sparvu     63M   14M sleep   59    0   0:04:11 1.3% realplay.bin/1
  6765 sparvu    129M   77M sleep   59    0   0:04:26 0.3% firefox-bin/3
  1922 root     5752K 4544K sleep   59    0   0:20:49 0.2% intrd/1
  7068 sparvu    249M  134M sleep   49    0   0:02:43 0.1% soffice.bin/5
  3291 sparvu     44M 7836K sleep   59    0   0:01:53 0.1% at-spi-registry/1
  3154 sparvu     50M   12M sleep   59    0   0:01:32 0.0% gnome-netstatus/1
  1967 root      102M   36M sleep   29   10   0:01:14 0.0% webservd/31
  1984 webservd  142M   54M sleep   59    0   0:01:11 0.0% webservd/76
 23319 sparvu   3416K 2872K cpu0    59    0   0:00:00 0.0% prstat/1
  1810 noaccess  177M   65M sleep   59    0   0:00:56 0.0% java/28
  3133 sparvu     49M   14M sleep   59    0   0:00:46 0.0% metacity/1
  3152 sparvu     51M   14M sleep   59    0   0:00:31 0.0% wnck-applet/1
 27399 sparvu     71M   36M sleep   37    4   0:00:13 0.0% gimp-2.0/1
  3156 sparvu     48M   11M sleep   59    0   0:00:25 0.0% mixer_applet2/1
  6983 sparvu   1204K  908K sleep   59    0   0:00:00 0.0% ksh/1
  3137 sparvu     56M   19M sleep   59    0   0:00:31 0.0% gnome-panel/1
  3111 sparvu   6052K 3036K sleep   59    0   0:00:06 0.0% xscreensaver/1
  3116 sparvu   8148K 3740K sleep   59    0   0:00:05 0.0% gnome-smproxy/1
   360 root     4660K 1848K sleep   59    0   0:00:00 0.0% automountd/2
   480 root     1736K  544K sleep   59    0   0:00:00 0.0% smcboot/1
   478 root     1740K  944K sleep   59    0   0:00:00 0.0% smcboot/1
 NPROC USERNAME  SIZE   RSS MEMORY      TIME  CPU
    57 sparvu   1491M  543M    27%   0:26:49  10%
    39 root      560M  307M    15%   1:08:55 3.8%
     1 webservd  142M   54M   2.6%   0:01:11 0.0%
     1 noaccess  177M   65M   3.2%   0:00:56 0.0%
     1 smmsp    6872K 1480K   0.1%   0:00:00 0.0%
     1 lp       2936K 1084K   0.1%   0:00:00 0.0%
     4 daemon     11M 6004K   0.3%   0:00:00 0.0%
Total: 104 processes, 371 lwps, load averages: 1.36, 1.30, 0.96
```

2.High CPU Utilisation, cont.

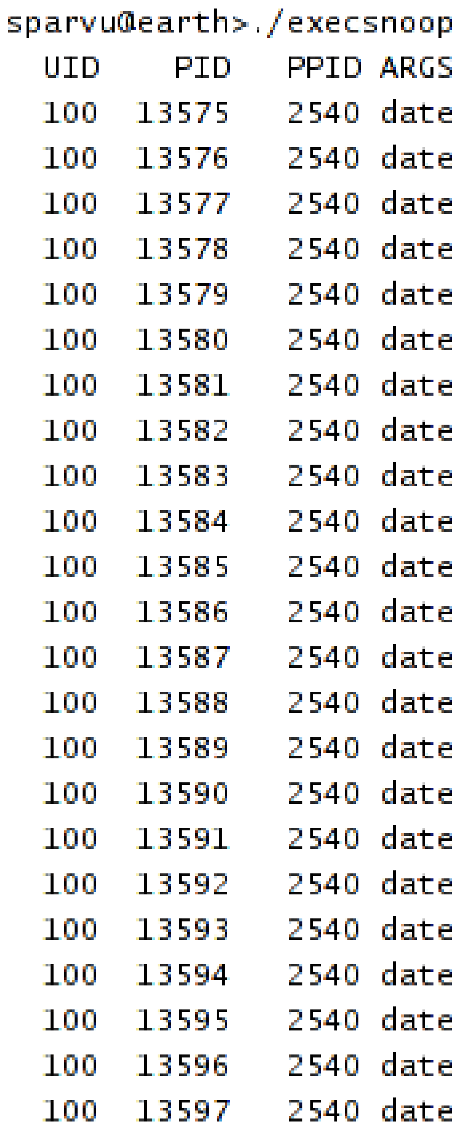
- Run *topsysproc*:



```
2006 May 28 17:43:08, load average: 0.56, 0.22, 0.12   syscalls: 46333

   PROCESS                              COUNT
   gnome-vfs-daemon                         3
   httpd                                    3
   mixer_applet2                            8
   xscreensaver                             9
   gnome-netstatus-                        10
   intrd                                   15
   java                                    20
   gnome-panel                             31
   mpstat                                  35
   tput                                    49
   webservd                                49
   dtrace                                  62
   firefox-bin                             84
   soffice.bin                            108
   sh                                     122
   clear                                  136
   Xorg                                   628
   gnome-terminal                        2455
   ksh                                   9727
   date                                 32778
```

2.High CPU Utilisation, cont.

- Run *execsnoop*:



```
sparvu@earth>./execsnoop
  UID    PID   PPID ARGS
  100  13575   2540 date
  100  13576   2540 date
  100  13577   2540 date
  100  13578   2540 date
  100  13579   2540 date
  100  13580   2540 date
  100  13581   2540 date
  100  13582   2540 date
  100  13583   2540 date
  100  13584   2540 date
  100  13585   2540 date
  100  13586   2540 date
  100  13587   2540 date
  100  13588   2540 date
  100  13589   2540 date
  100  13590   2540 date
  100  13591   2540 date
  100  13592   2540 date
  100  13593   2540 date
  100  13594   2540 date
  100  13595   2540 date
  100  13596   2540 date
  100  13597   2540 date
```

2.High CPU Utilisation, cont.

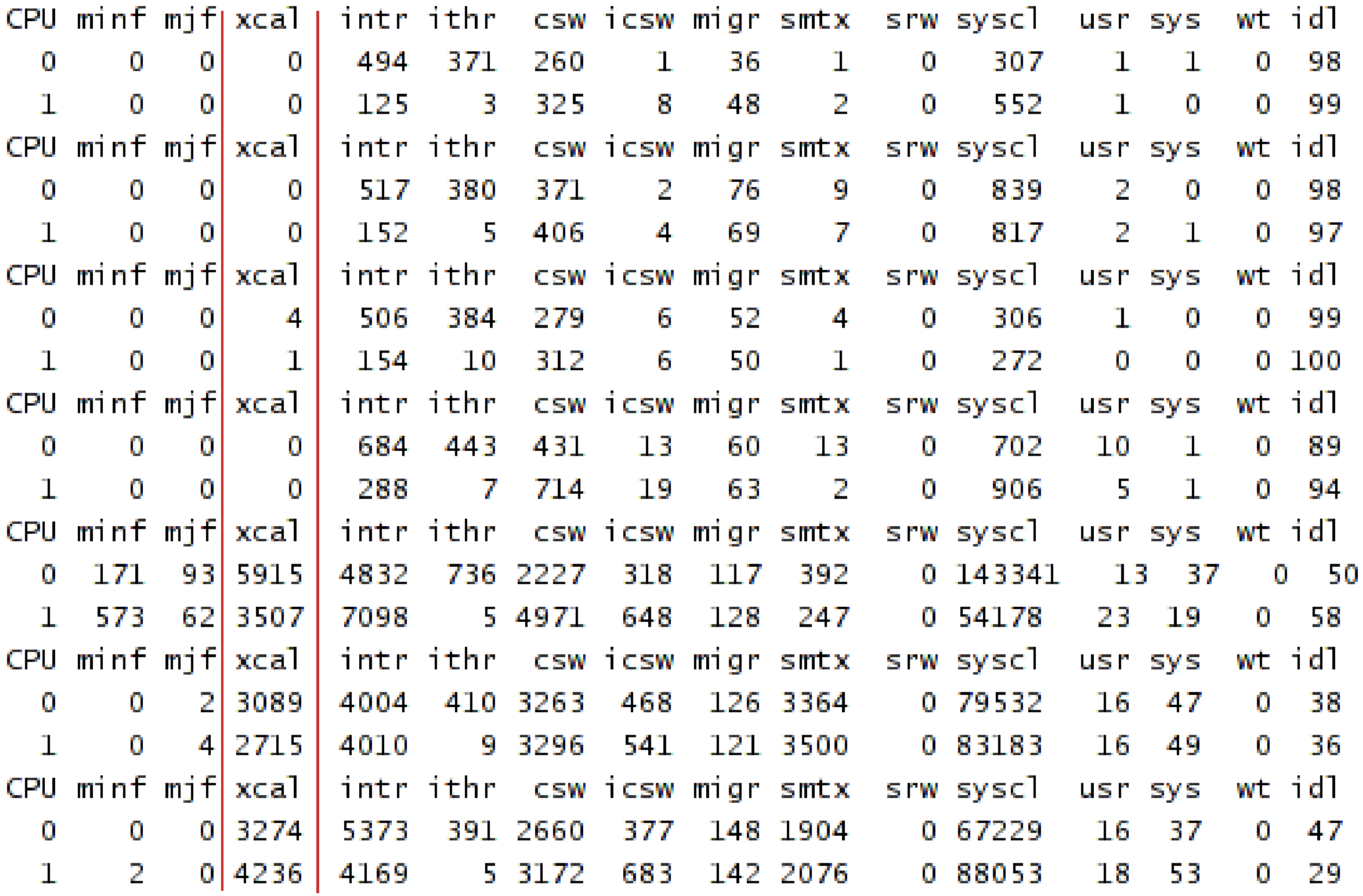
- **Conclusions:**
  - A high CPU utilisation was detected by *vmstat* and *prstat*. However the CPU consumption was not easy related to any process on the system
  - Using DTT utilities: *topsysproc* and *execsnoop* the real problem was very easily found and the process/owner generating all the load was easy identified

3.High Cross-Calls

- It has been detected on a multiprocessor server a high number of inter-processor cross-calls per second. This was discovered using *mpstat*
- Inter-processor cross-calls is a number indicating how often CPUs are sending the work from one to another. A clear indication of overhead
- Investigate using *mpstat* and see if it is easy to find out who generates all these cross-calls
- Solve the problem by using: *xcallsbypid.d* from DTT Cpu category

3.High Cross-Calls, cont.

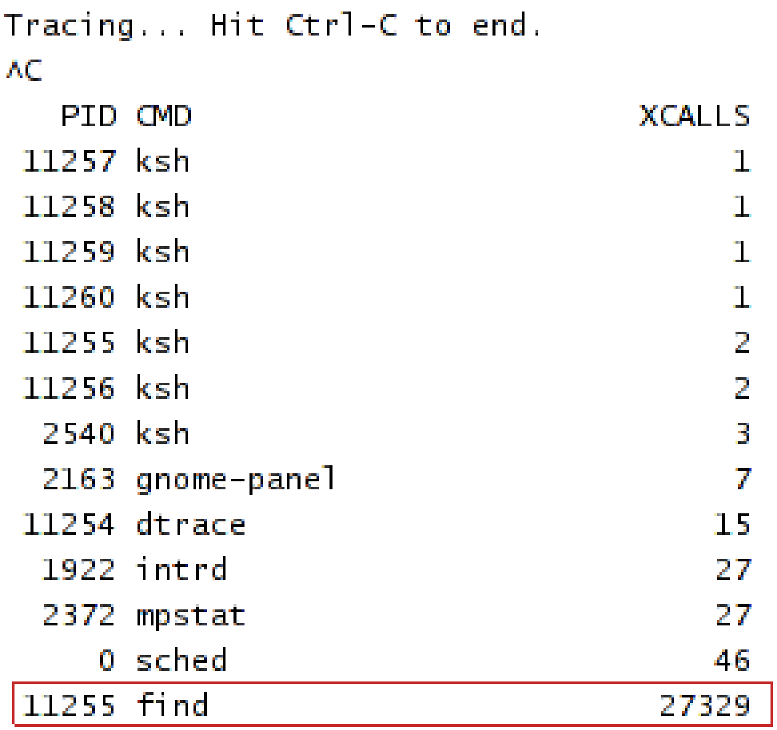
- *mpstat* reports:



```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   494  371  260    1   36    1    0   307    1   1   0  98
  1    0   0    0   125    3  325    8   48    2    0   552    1   0   0  99
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   517  380  371    2   76    9    0   839    2   0   0  98
  1    0   0    0   152    5  406    4   69    7    0   817    2   1   0  97
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    4   506  384  279    6   52    4    0   306    1   0   0  99
  1    0   0    1   154   10  312    6   50    1    0   272    0   0   0 100
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   684  443  431   13   60   13    0   702   10   1   0  89
  1    0   0    0   288    7  714   19   63    2    0   906    5   1   0  94
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0  171  93 5915  4832  736 2227  318  117  392    0 143341   13  37    0   50
  1  573  62 3507  7098    5 4971  648  128  247    0 54178   23  19   0  58
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   2 3089  4004  410 3263  468  126 3364    0 79532   16  47   0  38
  1    0   4 2715  4010    9 3296  541  121 3500    0 83183   16  49   0  36
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 3274  5373  391 2660  377  148 1904    0 67229   16  37   0  47
  1    2   0 4236  4169    5 3172  683  142 2076    0 88053   18  53   0  29
```

3.High Cross-Calls, cont.

- Run *xcallsbypid.d* from Cpu category:



```
Tracing... Hit Ctrl-C to end.
^C
   PID CMD                     XCALLS
 11257 ksh                          1
 11258 ksh                          1
 11259 ksh                          1
 11260 ksh                          1
 11255 ksh                          2
 11256 ksh                          2
  2540 ksh                          3
  2163 gnome-panel                  7
 11254 dtrace                      15
  1922 intrd                       27
  2372 mpstat                      27
     0 sched                       46
 11255 find                     27329
```

3.High Cross-Calls, cont.

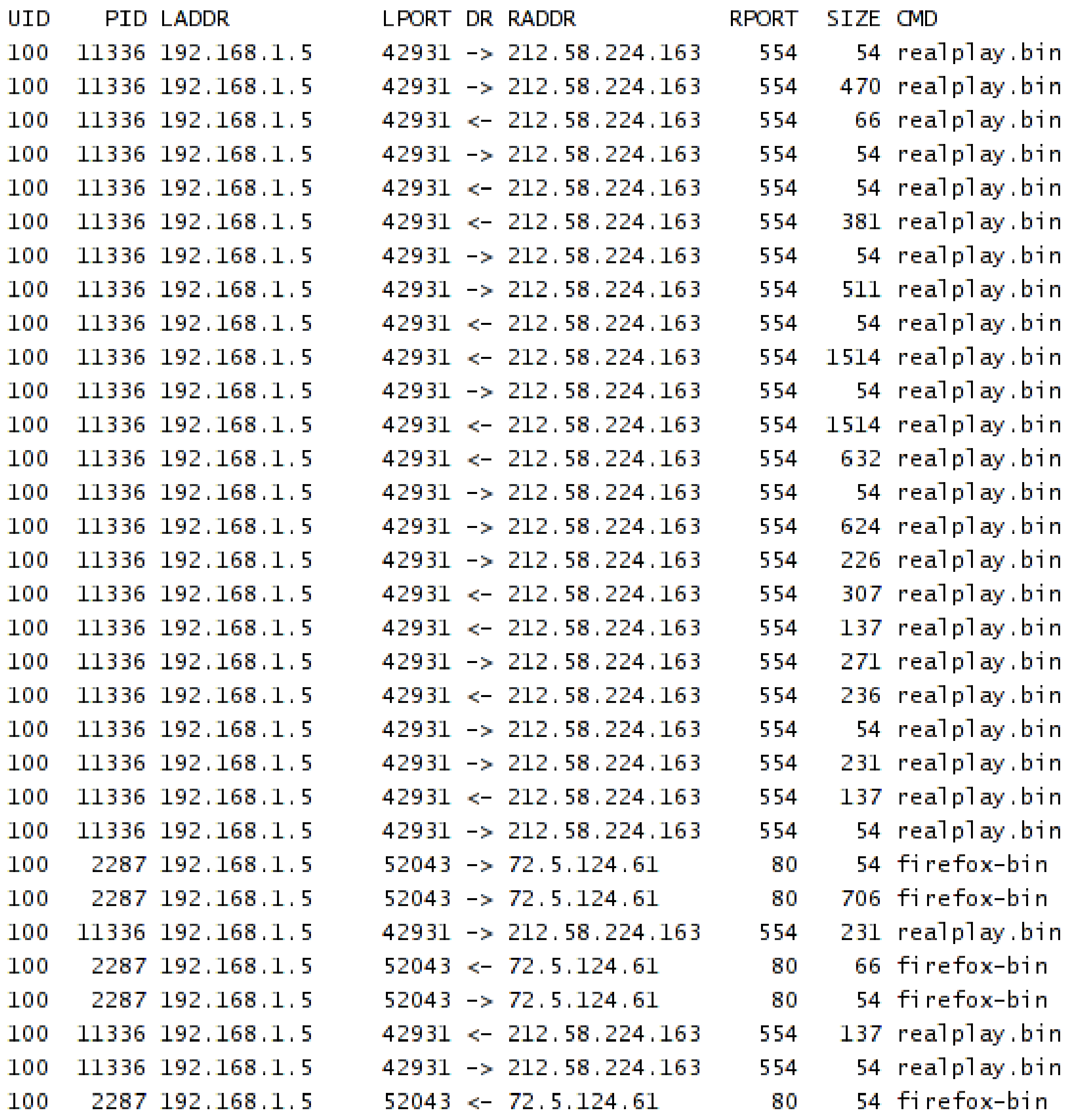
- **Conclusions:**
  - Solaris's *mpstat* was used to identify the high xcalls, however *mpstat* was not reporting on who was generating that big number
  - Very easy to identify the process/application which was generating lots of cross calls directly using DTT utility: *xcallsbypid.d*

4.Network Connections

- The network status utility *netstat* displays a status of all network connections on a system
- With the current tools there is no easy way to find out and co-relate a network connection with a process or the owner of it
- Extra tools like *lsof* can list what connections were made and by who
- What about incoming connections ?
- Solve the problem by using: *tcptop*, *tcpsnoop* and *connections* utilities from DTT

4.Network Connections, cont.

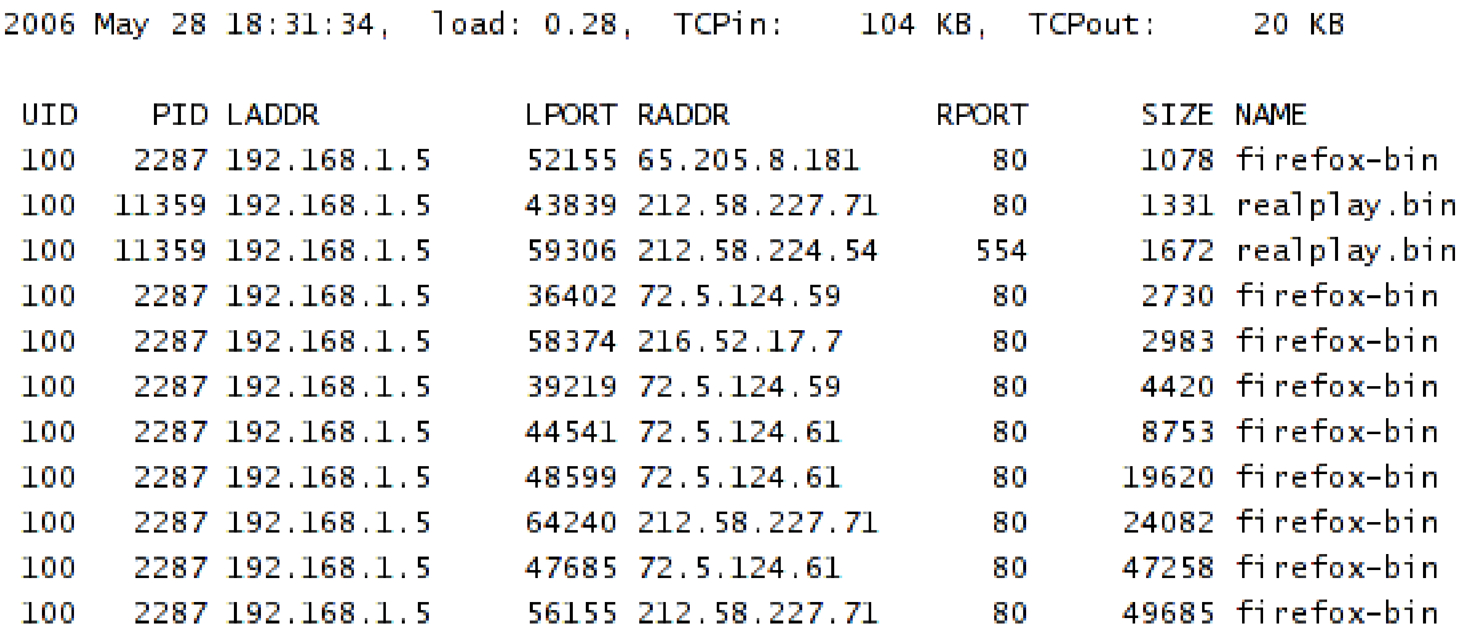
- Under Net category execute: *tcpsnoop*



```
UID    PID LADDR           LPORT DR RADDR           RPORT  SIZE CMD
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   470 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554    66 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   381 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   511 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554  1514 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554  1514 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   632 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   624 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   226 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   307 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   137 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   271 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   236 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   231 realplay.bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   137 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100   2287 192.168.1.5     52043 -> 72.5.124.61        80    54 firefox-bin
100   2287 192.168.1.5     52043 -> 72.5.124.61        80   706 firefox-bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554   231 realplay.bin
100   2287 192.168.1.5     52043 <- 72.5.124.61        80    66 firefox-bin
100   2287 192.168.1.5     52043 -> 72.5.124.61        80    54 firefox-bin
100  11336 192.168.1.5     42931 <- 212.58.224.163    554   137 realplay.bin
100  11336 192.168.1.5     42931 -> 212.58.224.163    554    54 realplay.bin
100   2287 192.168.1.5     52043 <- 72.5.124.61        80    54 firefox-bin
```

4.Network Connections, cont.

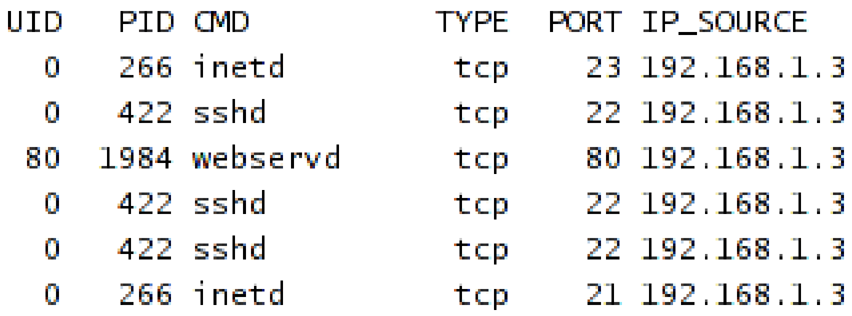
- To display top network packets run *tcptop*:



```
2006 May 28 18:31:34,  load: 0.28,  TCPin:    104 KB,  TCPout:     20 KB

 UID    PID LADDR           LPORT RADDR           RPORT      SIZE NAME
 100   2287 192.168.1.5     52155 65.205.8.181       80      1078 firefox-bin
 100  11359 192.168.1.5     43839 212.58.227.71      80      1331 realplay.bin
 100  11359 192.168.1.5     59306 212.58.224.54     554      1672 realplay.bin
 100   2287 192.168.1.5     36402 72.5.124.59        80      2730 firefox-bin
 100   2287 192.168.1.5     58374 216.52.17.7        80      2983 firefox-bin
 100   2287 192.168.1.5     39219 72.5.124.59        80      4420 firefox-bin
 100   2287 192.168.1.5     44541 72.5.124.61        80      8753 firefox-bin
 100   2287 192.168.1.5     48599 72.5.124.61        80     19620 firefox-bin
 100   2287 192.168.1.5     64240 212.58.227.71      80     24082 firefox-bin
 100   2287 192.168.1.5     47685 72.5.124.61        80     47258 firefox-bin
 100   2287 192.168.1.5     56155 212.58.227.71      80     49685 firefox-bin
```

4.Network Connections, cont.

- To monitor and check the incoming connections run *connections*:



```
UID    PID CMD            TYPE   PORT IP_SOURCE
  0    266 inetd           tcp     23 192.168.1.3
  0    422 sshd            tcp     22 192.168.1.3
 80   1984 webservd        tcp     80 192.168.1.3
  0    422 sshd            tcp     22 192.168.1.3
  0    422 sshd            tcp     22 192.168.1.3
  0    266 inetd           tcp     21 192.168.1.3
```

4.Network Connections, cont.

- **Conclusions:**
  - Not very easy to relate network connections to processes on the system or list the top of connections
  - Net category has a lot of scripts which can easily help like: *tcpsnoop*, *tcptop* and *connections*

5.Disk Utilization

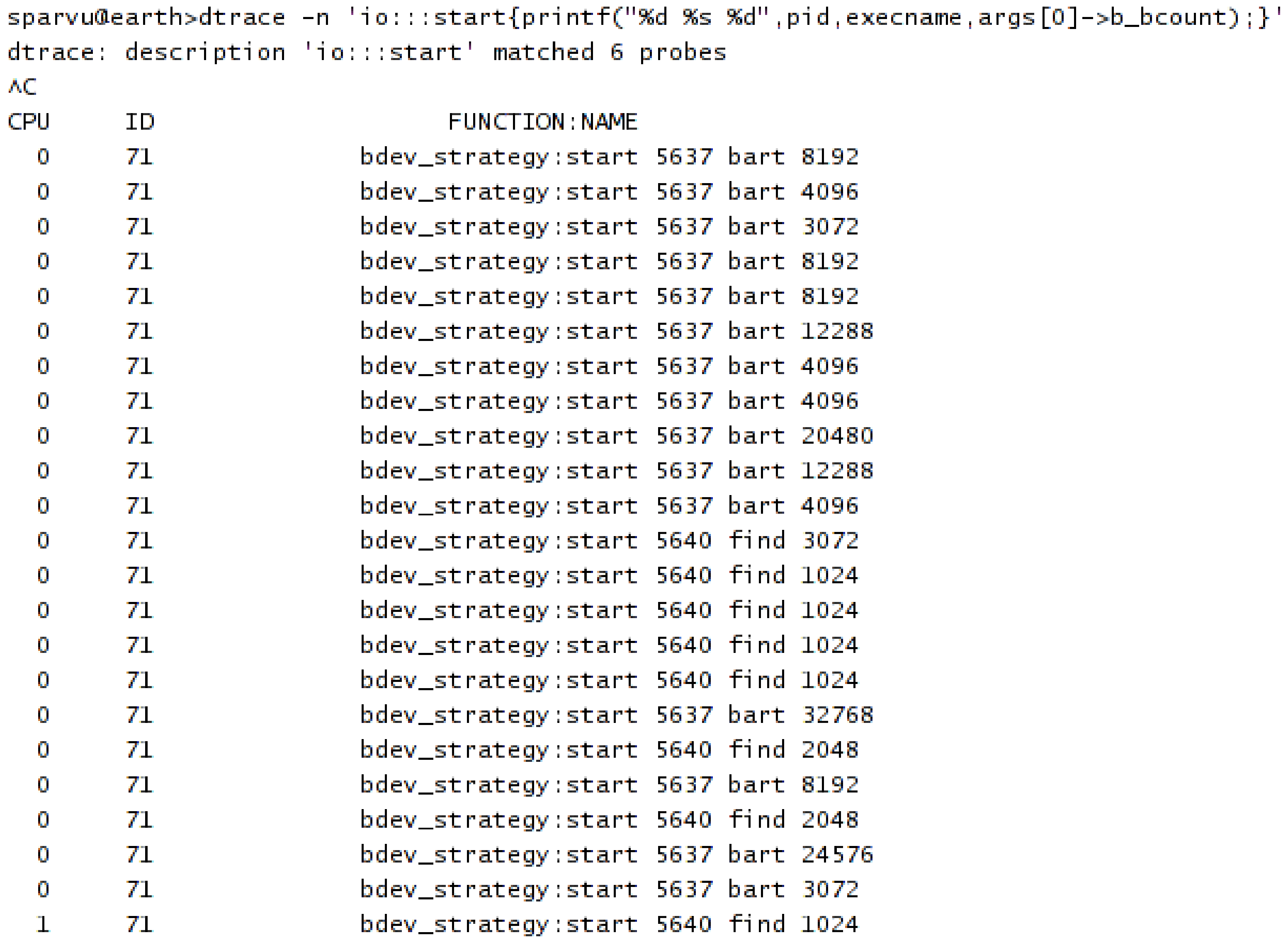
- Disk utilisation can be monitored using *iostat* – but to co-relate the utilisation with a process is a hard mission
- There are tools to check CPU usage by process but there are no tools to check disk I/O by process
- The old good friend: *iostat -xnmp*
- I/O type: reading *iostat* data a SysAdmin can describe if the I/O is sequential or random

5.Disk Utilization, cont.

- It is important to know what type of I/O there is: sequential or random
- How can you list what processes are generating I/O, or list disk events or how much a process is using the disk (size of the disk event or the service time of the disk events) ?
- Easily use the following DTT scripts: *iotop*, *iosnoop* from DTT root directory

5.Disk Utilization, cont.

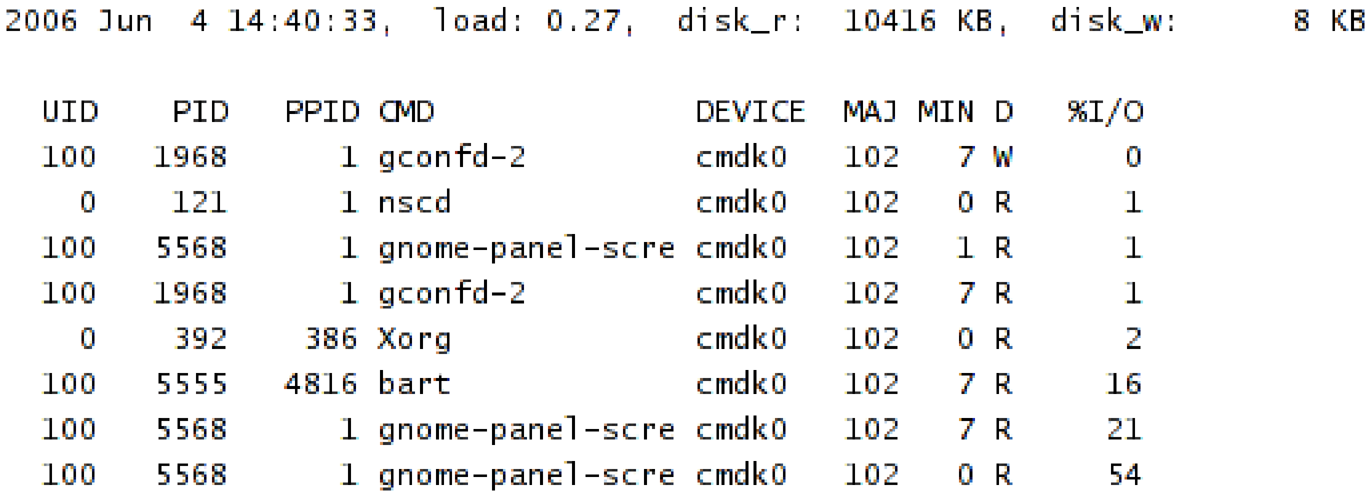
- One Liner says:



```
sparvu@earth>dtrace -n 'io:::start{printf("%d %s %d",pid,execname,args[0]->b_bcount);}'
dtrace: description 'io:::start' matched 6 probes
^C
CPU     ID                    FUNCTION:NAME
  0     71            bdev_strategy:start 5637 bart 8192
  0     71            bdev_strategy:start 5637 bart 4096
  0     71            bdev_strategy:start 5637 bart 3072
  0     71            bdev_strategy:start 5637 bart 8192
  0     71            bdev_strategy:start 5637 bart 8192
  0     71            bdev_strategy:start 5637 bart 12288
  0     71            bdev_strategy:start 5637 bart 4096
  0     71            bdev_strategy:start 5637 bart 4096
  0     71            bdev_strategy:start 5637 bart 20480
  0     71            bdev_strategy:start 5637 bart 12288
  0     71            bdev_strategy:start 5637 bart 4096
  0     71            bdev_strategy:start 5640 find 3072
  0     71            bdev_strategy:start 5640 find 1024
  0     71            bdev_strategy:start 5640 find 1024
  0     71            bdev_strategy:start 5640 find 1024
  0     71            bdev_strategy:start 5640 find 1024
  0     71            bdev_strategy:start 5637 bart 32768
  0     71            bdev_strategy:start 5640 find 2048
  0     71            bdev_strategy:start 5637 bart 8192
  0     71            bdev_strategy:start 5640 find 2048
  0     71            bdev_strategy:start 5637 bart 24576
  0     71            bdev_strategy:start 5637 bart 3072
  1     71            bdev_strategy:start 5640 find 1024
```

5.Disk Utilization, cont.

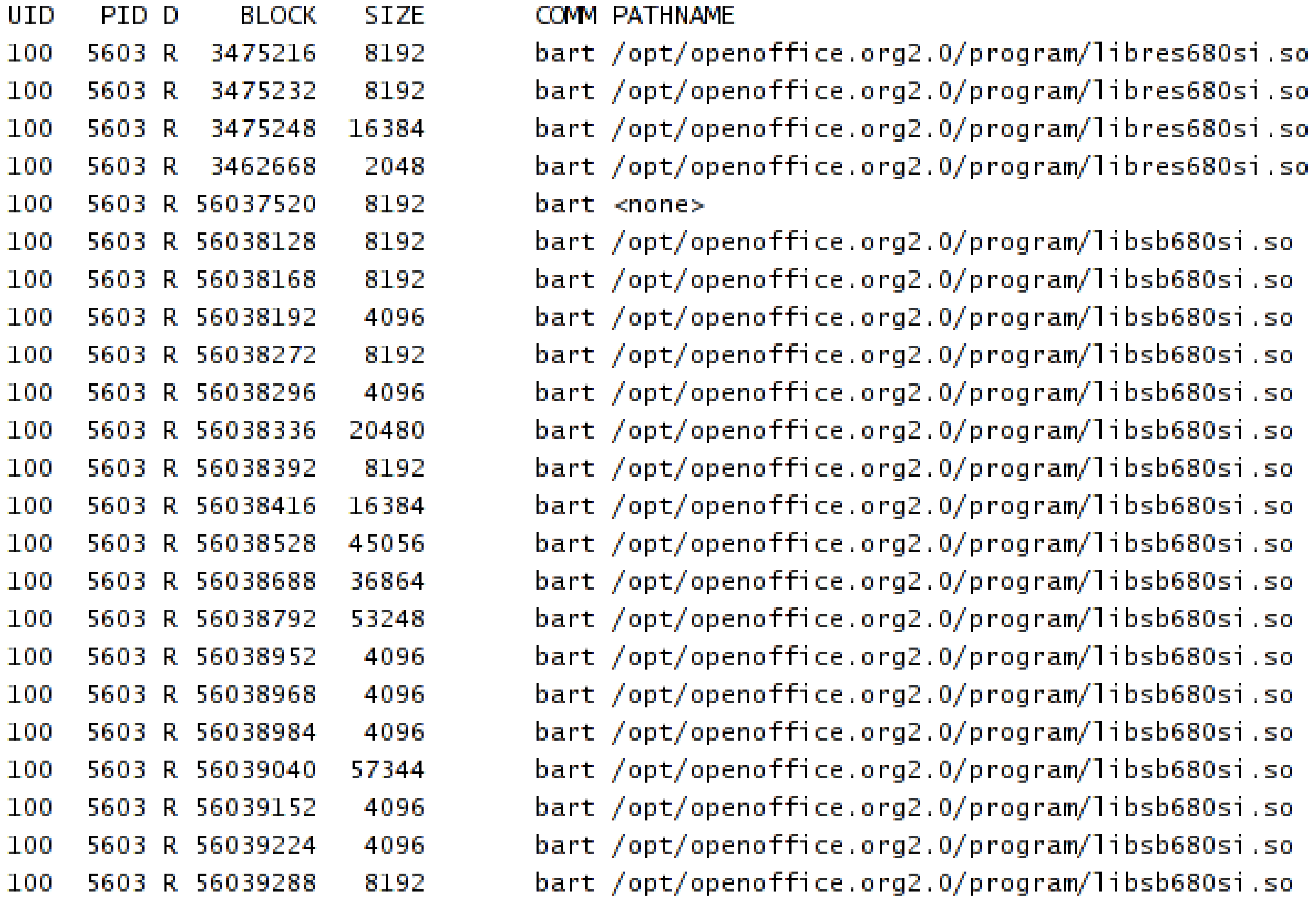
- Run *iotop*:



```
2006 Jun  4 14:40:33,  load: 0.27,  disk_r:  10416 KB,  disk_w:      8 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D   %I/O
  100   1968      1 gconfd-2         cmdk0   102   7 W      0
    0    121      1 nscd             cmdk0   102   0 R      1
  100   5568      1 gnome-panel-scre cmdk0   102   1 R      1
  100   1968      1 gconfd-2         cmdk0   102   7 R      1
    0    392    386 Xorg             cmdk0   102   0 R      2
  100   5555   4816 bart             cmdk0   102   7 R     16
  100   5568      1 gnome-panel-scre cmdk0   102   7 R     21
  100   5568      1 gnome-panel-scre cmdk0   102   0 R     54
```

5.Disk Utilization, cont.

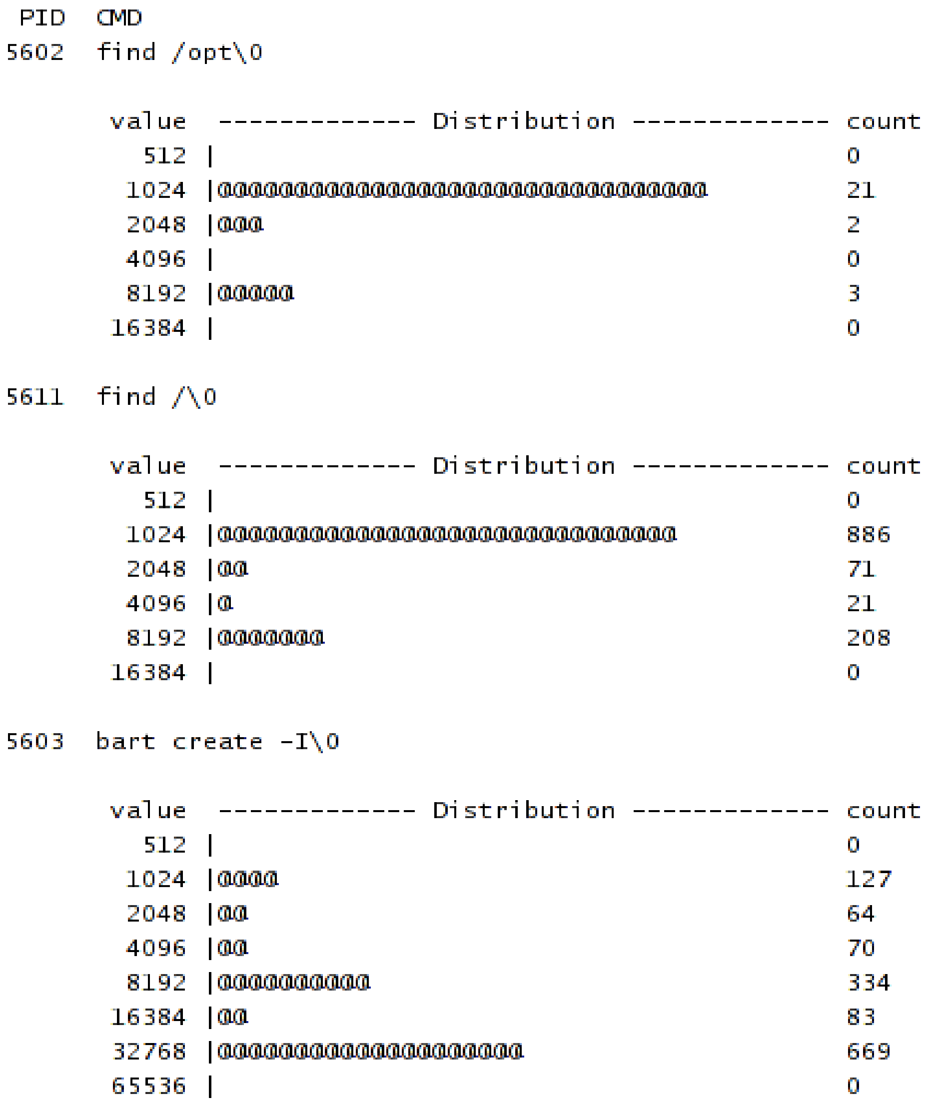
- Run now *iosnoop*:



```
UID   PID D    BLOCK   SIZE       COMM PATHNAME
100  5603 R  3475216   8192       bart /opt/openoffice.org2.0/program/libres680si.so
100  5603 R  3475232   8192       bart /opt/openoffice.org2.0/program/libres680si.so
100  5603 R  3475248  16384       bart /opt/openoffice.org2.0/program/libres680si.so
100  5603 R  3462668   2048       bart /opt/openoffice.org2.0/program/libres680si.so
100  5603 R 56037520   8192       bart <none>
100  5603 R 56038128   8192       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038168   8192       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038192   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038272   8192       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038296   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038336  20480       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038392   8192       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038416  16384       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038528  45056       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038688  36864       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038792  53248       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038952   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038968   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56038984   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56039040  57344       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56039152   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56039224   4096       bart /opt/openoffice.org2.0/program/libsb680si.so
100  5603 R 56039288   8192       bart /opt/openoffice.org2.0/program/libsb680si.so
```

5.Disk Utilization, cont.

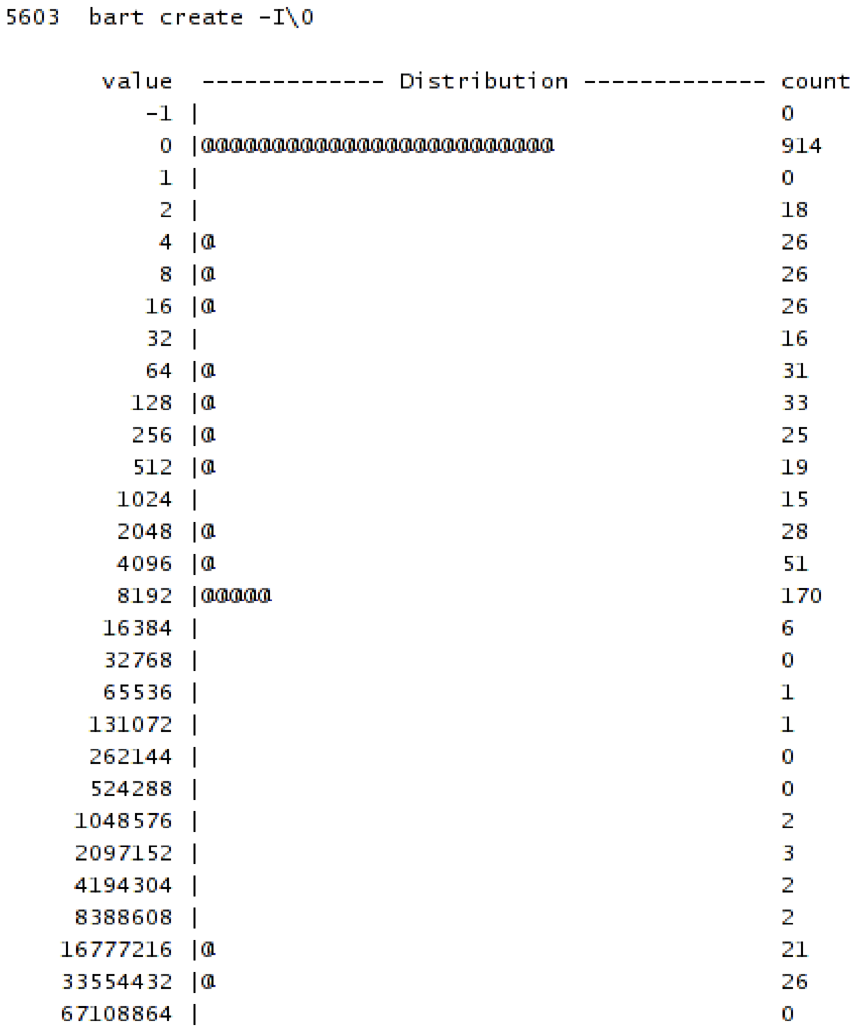
- How much the process reads...use *bitesize.d*:



```
 PID   CMD
5602   find /opt\0

        value  ------------- Distribution ------------- count
          512 |                                         0
         1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         21
         2048 |@@@                                      2
         4096 |                                         0
         8192 |@@@@@                                    3
        16384 |                                         0

5611   find /\0

        value  ------------- Distribution ------------- count
          512 |                                         0
         1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@           886
         2048 |@@                                       71
         4096 |@                                        21
         8192 |@@@@@@@                                  208
        16384 |                                         0

5603   bart create -I\0

        value  ------------- Distribution ------------- count
          512 |                                         0
         1024 |@@@@                                     127
         2048 |@@                                       64
         4096 |@@                                       70
         8192 |@@@@@@@@@@                               334
        16384 |@@                                       83
        32768 |@@@@@@@@@@@@@@@@@@@@                     669
        65536 |                                         0
```

5.Disk Utilization, cont.

- Look for seek distance of the disk events. Run *seeksize.d* to understand if the I/O is sequential or not:



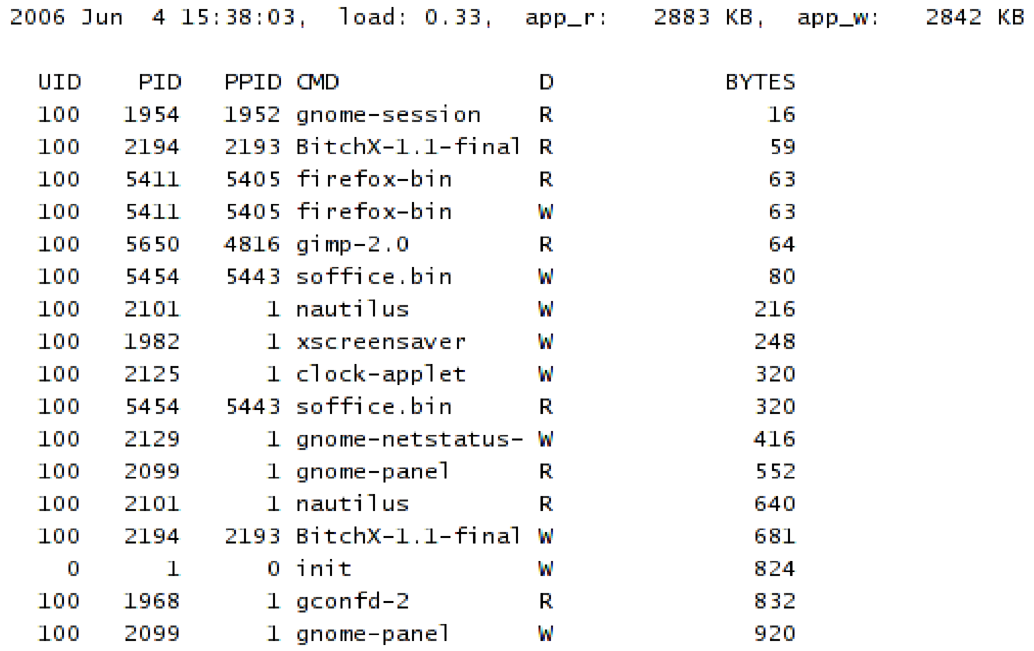
```
  5603  bart create -I\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@                914
               1 |                                         0
               2 |                                         18
               4 |@                                        26
               8 |@                                        26
              16 |@                                        26
              32 |                                         16
              64 |@                                        31
             128 |@                                        33
             256 |@                                        25
             512 |@                                        19
            1024 |                                         15
            2048 |@                                        28
            4096 |@                                        51
            8192 |@@@@@                                    170
           16384 |                                         6
           32768 |                                         0
           65536 |                                         1
          131072 |                                         1
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         2
         2097152 |                                         3
         4194304 |                                         2
         8388608 |                                         2
        16777216 |@                                        21
        33554432 |@                                        26
        67108864 |                                         0
```

5.Disk Utilization, cont.

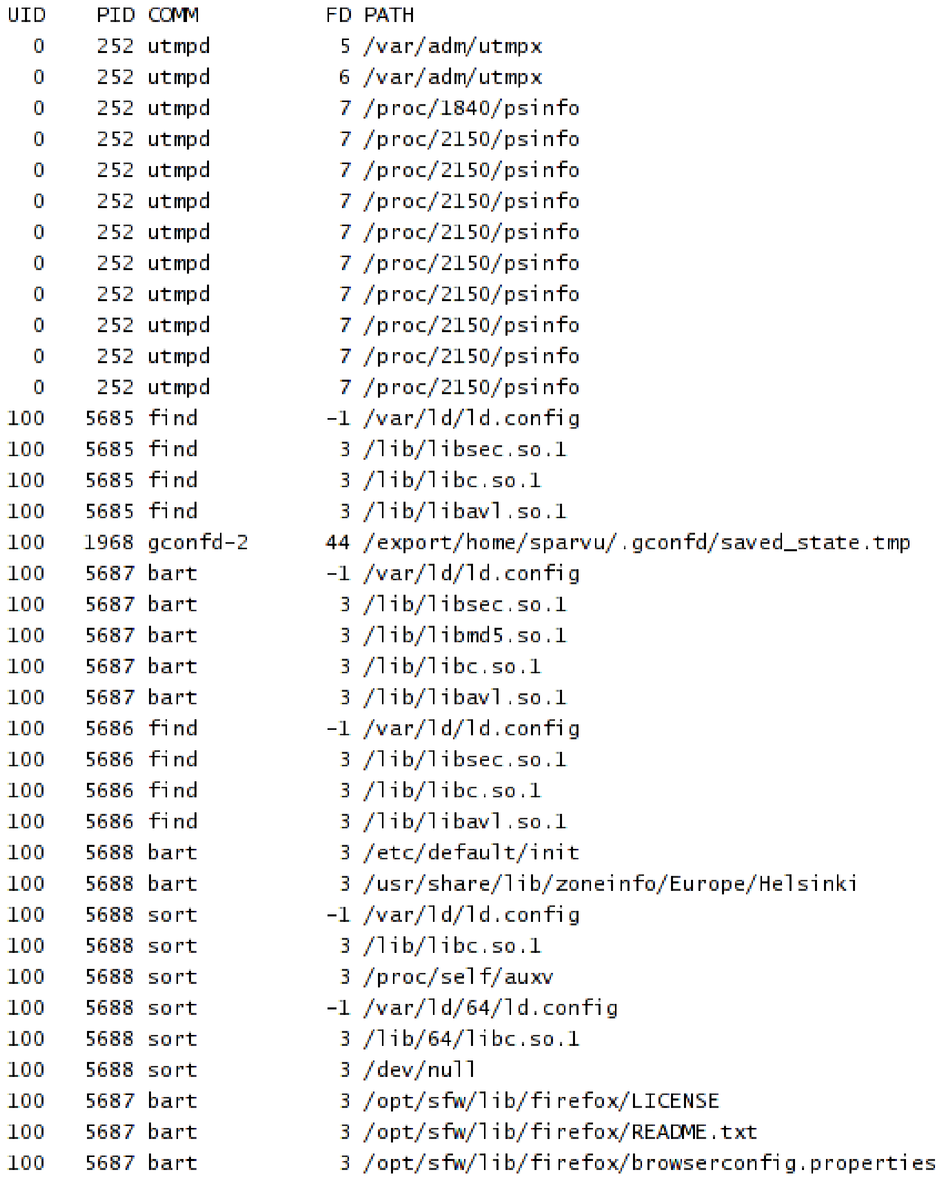
- Other important DTT utilities used to measure and analyse disk I/O events
- *rwsnoop*: snoops the read/write operations
- *rwtop*: used to display the top read/write operations by process id
- *opensnoop*: used to snoop what files are being open and by who. Very easy to discover what processes are opening what files

5.Disk Utilization, cont.

- *rwtop* and *opensnoop*:



```
2006 Jun  4 15:38:03,  load: 0.33,  app_r:   2883 KB,  app_w:   2842 KB

  UID    PID   PPID CMD              D            BYTES
  100   1954   1952 gnome-session    R               16
  100   2194   2193 BitchX-1.1-final R               59
  100   5411   5405 firefox-bin      R               63
  100   5411   5405 firefox-bin      W               63
  100   5650   4816 gimp-2.0         R               64
  100   5454   5443 soffice.bin      W               80
  100   2101      1 nautilus         W              216
  100   1982      1 xscreensaver     W              248
  100   2125      1 clock-applet     W              320
  100   5454   5443 soffice.bin      R              320
  100   2129      1 gnome-netstatus- W              416
  100   2099      1 gnome-panel      R              552
  100   2101      1 nautilus         R              640
  100   2194   2193 BitchX-1.1-final W              681
    0      1      0 init             W              824
  100   1968      1 gconfd-2         R              832
  100   2099      1 gnome-panel      W              920
```



```
UID    PID COMM          FD PATH
  0    252 utmpd          5 /var/adm/utmpx
  0    252 utmpd          6 /var/adm/utmpx
  0    252 utmpd          7 /proc/1840/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
  0    252 utmpd          7 /proc/2150/psinfo
100   5685 find          -1 /var/ld/ld.config
100   5685 find           3 /lib/libsec.so.1
100   5685 find           3 /lib/libc.so.1
100   5685 find           3 /lib/libavl.so.1
100   1968 gconfd-2      44 /export/home/sparvu/.gconfd/saved_state.tmp
100   5687 bart          -1 /var/ld/ld.config
100   5687 bart           3 /lib/libsec.so.1
100   5687 bart           3 /lib/libmd5.so.1
100   5687 bart           3 /lib/libc.so.1
100   5687 bart           3 /lib/libavl.so.1
100   5686 find          -1 /var/ld/ld.config
100   5686 find           3 /lib/libsec.so.1
100   5686 find           3 /lib/libc.so.1
100   5686 find           3 /lib/libavl.so.1
100   5688 bart           3 /etc/default/init
100   5688 bart           3 /usr/share/lib/zoneinfo/Europe/Helsinki
100   5688 sort          -1 /var/ld/ld.config
100   5688 sort           3 /lib/libc.so.1
100   5688 sort           3 /proc/self/auxv
100   5688 sort          -1 /var/ld/64/ld.config
100   5688 sort           3 /lib/64/libc.so.1
100   5688 sort           3 /dev/null
100   5687 bart           3 /opt/sfw/lib/firefox/LICENSE
100   5687 bart           3 /opt/sfw/lib/firefox/README.txt
100   5687 bart           3 /opt/sfw/lib/firefox/browserconfig.properties
```

DTrace & Java

DTrace and Java

- DTrace can be used to debug and observe Java applications
- Easy to start: use *jstack()*, to display the Java activity as a stack backtrace. *jstack()* based on *ustack()*
- Useful to understand the I/O and scheduling caused by your Java application
- Java 5: VM agents, shared libraries which are dynamically loaded when the VM starts
- Java 6, Mustang, introduces two new providers: hotspot and hotspot_jni

DTrace and Java, cont.

- *jstack()*
- The simplest form to record a stack trace from a Java application
  - Not jstackstrsize default of 512 may need to be increased

```
dtrace -x jstackstrsize=1k -n syscall: ...
```

- Delivered already with DTrace framework:

```
$ dtrace -n 'syscall::entry/pid==xxx/
  {jstack(40);}'


$ dtrace -n 'syscall::entry/pid==xxx/
  {@[jstack(40)] = count();}'
```

jstack() Action

- jstack action prints mixed mode stack trace
- Both java frames and native (C/C++) frames are shown
- Only JVM versions 5.0_01 and later are supported
- jstack shows hex numbers for JVM versions before 5.0_01

```
#!/usr/sbin/dtrace -s
syscall::pollsys:entry
/ pid == $1 /   {
    jstack(50,8192);
}
```

- first optional argument limits the number of frames shown
- second optional argument changes the string size
- jstackstrsize pragma / -x to increase buffer for all jstack()'s

jstack()

```
libc.so.1`__pollsys+0x7
libc.so.1`pselect+0x19e
libc.so.1`select+0x69
libXt.so.4`IoWait+0x36
libXt.so.4`_XtWaitForSomething+0x1a9
libXt.so.4`XtAppPending+0x188
libmawt.so`0xd43d3928
libmawt.so`0xd43d37d6
libmawt.so`Java_sun_awt_motif_MToolkit_run+0x34
sun/awt/motif/MToolkit.run
java/lang/Thread.run
StubRoutines (1)
libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_pnMmethodHandle_pnRJavaCallArguments_pnGThread__v_+0x187
libjvm.so`__1cCosUos_exception_wrapper6FpFpnJJavaValue_pnMmethodHandle_pnRJavaCallArguments_pnGThread__v2468_v_+0x14
libjvm.so`__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle_pnRJavaCallArguments_pnGThread__v_+0x28
libjvm.so`__1cJJavaCallsMcall_virtual6FpnJJavaValue_nLKlassHandle_nMsymbolHandle_4pnRJavaCallArguments_pnGThread__v_+0xbe
libjvm.so`__1cJJavaCallsMcall_virtual6FpnJJavaValue_nGHandle_nLKlassHandle_nMsymbolHandle_5pnGThread__v_+0x6d
libjvm.so`__1cMthread_entry6FpnKJavaThread_pnGThread__v_+0xd0
libjvm.so`__1cKJavaThreadRthread_main_inner6M_v_+0x51
libjvm.so`__1cKJavaThreadDrun6M_v_+0x105
libjvm.so`__1cG_start6Fpv_0_+0xd2
libc.so.1`_thr_setup+0x51
libc.so.1`_lwp_start
397

libc.so.1`stat64+0x7
java/io/UnixFileSystem.getBooleanAttributes0*
0x20245c8b
932
```

The dvm Provider

- java.net project to add DTrace support in
  - 1.4.2 (libdvmpi.so)
  - 1.5 (libdvmti.so)
  - https://solaris10-dtrace-vm-agents.dev.java.net/
- Download shared libs
- Add location of libs to LD_LIBRARY_PATH variable
- Set JAVA_TOOL_OPTIONS to -Xrundvmti:all
- Name of provider - “dvm”

The dvm Provider: Probes

- dvm probes and their signatures

  vm-init(), vm-death()

  thread-start(char *thread_name), thread-end()

  class-load(char *class_name)

  class-unload(char *class_name)

  gc-start(), gc-finish()

  gc-stats(long used_objects, long used_object_space)

  object-alloc(char *class_name, long size)

  object-free(char *class_name)

  method-entry(char *class_name, char *method_name, char *method_signature)

  method__return(char *class_name, char *method_name, char *method_signature)

The dvm Provider: alloc and free

- Object allocation/deallocation

```
#!/usr/sbin/dtrace -qs
dvm$target:::object-alloc
{
  printf("%s allocated %d size objects\n”,
    copyinstr(arg0), arg1);
}

dvm$target:::object-free
{
  printf("%s freed %d size objects\n",
    copyinstr(arg0), arg1);
}

# ./java_alloc.d -p `pgrep -n java`
```

The dvm Provider: Methods

- Count methods called

```
#!/usr/sbin/dtrace -s

dvm$target:::method-entry
{
  @[copyinstr(arg0),copyinstr(arg1)] = count();
}

# ./java_method_count.d -p `pgrep -n java`
```

The dvm provider: Time Spent

- Time spent in methods

```
#!/usr/sbin/dtrace -s
dvm$target:::method-entry
{
  self->ts[copyinstr(arg0),copyinstr(arg1)] =
    vtimestamp;
}

dvm$target:::method-return
{
  @ts[copyinstr(arg0),copyinstr(arg1)] =
    sum(vtimestamp - self->ts[copyinstr(arg0),
      copyinstr(arg1)]);
}
```

```
# ./java_method.d -p `pgrep -n java`
```

DTrace and Java, cont.

- VM Agents
  - Some probes have a significant probe effect, and require enabling when the JVM is started

    -XX:+ExtendedDtraceProbes

    jinfo -XX:+ExtendedDtraceProbes

DTrace and Java, cont.

- Java 6, Mustang
  - Added two new providers: hotspot and hotspot_jni
  - Using these providers it is now possible to collect data from your Java applications
  - Hotspot_jni: probes related with Java Native Interface
  - Hotspot provider:

    VM Probes: Initialization and Shutdown
    Thread statistics Probes
    Class loading and unloading Probes
    Garbage Collection Probes
    Method Compilation Probes

DTrace in JDK 6

- hotspot provider implements all dvm probes plus extensions:
  - Method compilation (method-compile-begin/end)
  - Compiled method load/unload(compiled-method-load/unload)
  - JNI method probes.
    - DTrace probes as entry and return from each JNI method.
- Strings are now unterminated UTF-8 data. Always use associated length value with copyinstr().

Method Compilation Probes

```
hotspot$1:::method-compile-begin {
    self->str = (char*) copyin(arg2, arg3+1);
    self->str[arg3] = '\0';
    self->classname = (string)self->str;
    self->str = (char*) copyin(arg4, arg5+1);
    self->str[arg5] = '\0';
    self->methodname = (string)self->str;
    printf("Compile begin %s.%s\n",
      self->classname, self->methodname);
}
```

Exception Stack Trace

```
hotspot$1:::method-entry {
    self->ptr = (char*)copyin(arg1, arg2+1);
    self->ptr[arg2] = '\0';
    self->classname = (string)self->ptr;
    self->ptr = (char*)copyin(arg3, arg4+1);
    self->ptr[arg4] = '\0';
    self->methodname = (string)self->ptr;
}
hotspot$1:::method-entry
/self->classname == "java/lang/Throwable" &&
   self->methodname == "<init>"/
{
   jstack();
}
```

JDK 6 DTrace Usage

- Certain probes are expensive
  - Turned off by default
  - object-alloc
  - method-entry, method-return
  - monitor probes
    - monitor-wait, monitor-contended-enter, etc
- Requires you to start your application with the flag
  `-XX:+ExtendedDTraceProbes`
- Use -XX:+DTrace{Alloc,Method,Monitor}Probes if possible

JDK6 hotspot_jni Provider

- Probes for Java Native Interface (JNI)
- Located at entry/return points of all JNI functions
- Probe arguments are same as corresponding JNI function arguments (for _entry probes)
- For XXX_return probes, probe argument is return value
- Examples:

```
hotspot_jni$1:::GetPrimitiveArrayCritical_entry
hotspot_jni$1:::GetPrimitiveArrayCritical_return
```

JDK 1.6 and DTrace

- Check out `/usr/jdk/jdk1.6.0_06/sample/dtrace`

class_loading_stat.d The script collects statistics about loaded and unloaded Java classes and dump current state to stdout every N seconds.

gc_time_stat.d The script measures the duration of a time spent in GC. The duration is measured for every memory pool every N seconds.

hotspot_calls_tree.d The script prints calls tree of fired 'hotspot' probes.

method_compile_stat.d The script prints statistics about N methods with largest/smallest compilation time every M seconds.

method_invocation_stat.d The script collects statistics about Java method invocations.

method_invocation_stat_filter.d The script collects statistics about Java method invocations. You can specify package, class or method name to trace.

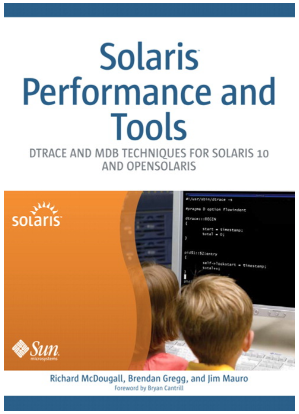
method_invocation_tree.d The script prints tree of Java and JNI method invocations.

monitors.d The script traces monitor related probes.

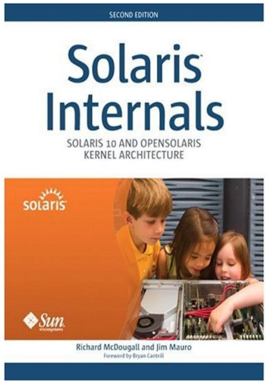
object_allocation_stat.d The script collects statistics about N object allocations every M seconds.

DTrace Community, cont.

- Solaris Internals 2nd
  - an update to Solaris Internals, for Solaris 10 and OpenSolaris. It covers Virtual Memory, File systems, Zones, Resource Management, Process Rights etc (all the good stuff in S10). This book is about 1100 pages
- New Solaris Performance and Tools !
  - aimed at Administrators to learn about performance and debugging. It's basically the book to read to understand and learn DTrace, MDB and the Solaris Performance tools, and a methodology for performance observability and debugging. This book is about 550 pages



Solaris Performance and Tools
DTRACE AND MDB TECHNIQUES FOR SOLARIS 10 AND OPENSOLARIS
Richard McDougall, Brendan Gregg, and Jim Mauro
Foreword by Bryan Cantrill



SECOND EDITION
Solaris Internals
SOLARIS 10 AND OPENSOLARIS KERNEL ARCHITECTURE
Richard McDougall and Jim Mauro
Foreword by Bryan Cantrill

DTrace Community, cont.

- Build around OpenSolaris community
- Available under www.opensolaris.org
  - The main page:

    http://www.opensolaris.org/os/community/dtrace/

    IRC on irc.freenode.net channels: #opensolaris, #dtrace
- The leaders:
  - Bryan M. Cantrill
  - Adam H. Leventhal
  - Mike Shapiro
  - Brendan Gregg
- Working with other communities

DTrace Community, cont.

- Jim Mauro and Richard McDougall: Solaris Internals
  - www.solarisinternals.com
- Lots of folks:
  - http://www.opensolaris.org/os/community/dtrace/observers/
- How can you help ? Use, Improve and Evangelize

Future

- Visualization tools
- Integration with Java 6
- New providers: Apache, Sun Java System Webserver
- DTrace and Zones: support already in Solaris Express builds
- Better documentation and more scripts
- DTrace and other operating systems:
  - FreeBSD: porting already done !
  - Linux: using SystemTap still experimental !

Database Supplement

- DTrace for Database Administrators
  - Learn how to use DTrace
  - Easy to use and experiment using DTraceToolkit
  - Understand how the entire database engine works
  - Special glasses: I/O monitoring
- **DTracing Oracle!!!**
- Real Case Examples

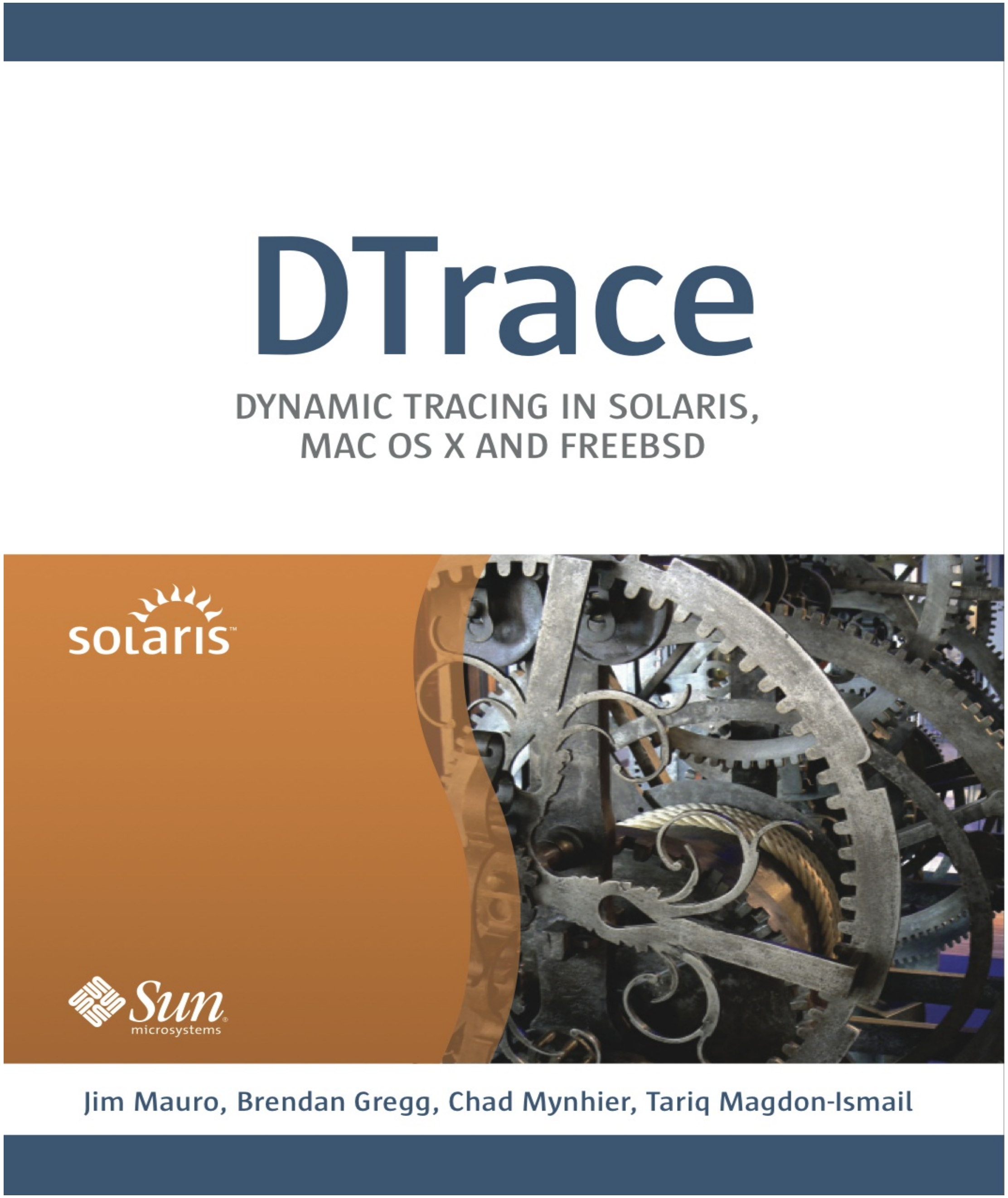
Free your mind

- A new mentality when debugging and observe with DTrace
- See the entire system
- Discover certain locations you want to investigate and look
- Place probes there, where are you interested
- Wait and see when the probes are executing
- Observe these locations by discovering who, how and when are accessed
- Gather the results by building a report

Free your mind, cont.

- Using DTrace does not mean you should not use anymore: vmstat, iostat, mpstat, etc.
- Try to understand every monitoring tool
- You don't have to do everything using DTrace...e.g.: memory leaks use the best tool: libumem, dbx
- Solaris has a very rich support for monitoring and observability. Try to understand each tool and what is good for: memory, disk, network, cpu, tracing, process monitoring and debug, kernel debug

Coming Soon!



DTrace

DYNAMIC TRACING IN SOLARIS,
MAC OS X AND FREEBSD

Jim Mauro, Brendan Gregg, Chad Mynhier, Tariq Magdon-Ismail open

USE IMPROVE EVANGELIZE

Thank you! (and to Jim Mauro et al.)

Harry J Foxwell, PhD
harry.foxwell@oracle.com

開
放
的
열린
مفتوح
libre
मुक्त
ಮುಕ್ತ
livre
libero
ముక్త
开放的
açık
open
nyílt
⠕⠏⠑⠝
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை

“open” artwork and icons by chandan:
http://blogs.sun.com/chandan