



[www.aulaunix.org](http://www.aulaunix.org)

# Introducción a DTrace

Iban Nieto Castillero, [blog.alucardx.net](http://blog.alucardx.net)

[www.alucardx.net](http://www.alucardx.net)

[iban.nieto@gmail.com](mailto:iban.nieto@gmail.com)

<OpenSolaris>

## Índice de contenidos

<b>1.- DTrace, preguntas y respuestas.....</b>	<b>4</b>
1.1.- ¿Qué es DTrace?.....	4
1.2.- ¿Para qué se utiliza DTrace?.....	4
1.3.- ¿Quién puede usar DTrace? .....	4
1.4.- ¿Es necesario tener conocimientos internos del kernel para emplear DTrace?.....	5
1.5.- ¿Qué privilegios necesito para utilizar DTrace?.....	5
1.6.- ¿Existe DTrace para otros sistemas operativos?.....	6
1.7.- ¿No se había inventado ya algo similar hace 20 años para los mainframes?.....	6
1.8.- ¿Hay libros sobre DTrace?.....	6
1.9.- ¿Existe algún caso de éxito empleando DTrace?.....	7
<b>2.- ¿Cómo funciona DTrace y el lenguaje D?.....</b>	<b>8</b>
2.1.- ¿Qué son los probes y providers?.....	9
<b>3.- Ejemplos trabajando con DTrace.....</b>	<b>10</b>
3.1.- Llamadas al sistema (syscalls).....	10
3.2.- Entrada/Salida de disco (Disk I/O).....	12
<b>4.- CHIME - un interfaz gráfico para DTrace.....</b>	<b>15</b>

## Licencia

Esta obra está bajo una licencia Reconocimiento-NoComercial-SinObraDerivada-2.5 España de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-nd/2.5/es> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Usted es libre de:

- Copiar, distribuir y comunicar públicamente la obra.

Bajo las condiciones siguientes:

- **Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciadore.
- **No comercial.** No puede utilizar esta obra para fines comerciales.
- **Sin obras derivadas.** No se puede alterar, transformar o generar una obra derivada a partir de esta obra.
- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.

Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

## Referencias

Todos los nombres propios de programas, sistemas operativos, equipos hardware, etc., que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

## 1.- DTrace: preguntas y respuestas

### 1.1.- ¿Qué es DTrace?

DTrace es una herramienta de depuración introducida en el Sistema Operativo Solaris 10 que nos puede ayudar a depurar problemas sistemáticos y/o difíciles de diagnosticar con las herramientas y mecanismos tradicionales. Nos ofrece una vista comprensible del comportamiento del sistema operativo y de las aplicaciones que se ejecutan sobre él. Con funciones similares a las de truss, apptrace, prex y mdb, DTrace integra en una única y poderosa herramienta de instrumentación que examina la actividad de usuario y del kernel de Solaris.

Está considerada actualmente como la única herramienta disponible que es lo suficientemente segura para utilizar en sistemas de producción, además con un insignificante impacto en el rendimiento (0% cuando no se emplea).

DTrace ha sido además el primer mayor componente del código de Solaris 10 que fue abierto a la comunidad opensource.

### 1.2.- ¿Para qué se utiliza DTrace?

Se utiliza para análisis de rendimiento, observación, búsqueda de problemas, depurado... Ejemplos como revisar los detalles de la Entrada/Salida de disco (Disk I/O) en tiempo real ó cronometrar las funciones de usuario para determinar si existe problemática alguna en el código.

### 1.3.- ¿Quién puede usar DTrace?

Primero necesitas ser root ó tener algún privilegio DTrace para poder emplear la herramienta.

- Los administradores de sistemas pueden utilizar DTrace para entender el comportamiento del sistema operativo y de las aplicaciones.

- Los programadores de aplicaciones puede utilizar DTrace para recoger tiempos y argumentar detalles de las funciones que ellos escriben, tanto en entornos de desarrollo como en sistemas en producción.

- Los ingenieros del kernel y drivers pueden emplear DTrace para depurar un kernel ya cargado en memoria así como todos sus módulos sin necesidad de ejecutar drivers en modo depuración (DEBUG).

#### 1.4.- ¿Es necesario tener conocimientos internos del kernel para emplear DTrace?

Cualquiera puede emplear DTrace desde el principio utilizando la documentación de los scripts ya escritos de [DTraceToolkit](#) ó los scripts de una única línea documentados en [Solaris Performance and Tools](#). Al principio no necesitarán escribir sus propios scripts, pero a medida que vayan surgiendo necesidades, los usuarios encontrarán que escribir sus propios scripts puede probar una mayor eficacia a la hora de resolver problemas en sus entornos.

No es necesario tener conocimientos del kernel para estudiar el código a nivel de usuario. Los desarrolladores de aplicaciones pueden estudiar las funciones que ellos mismos escriben y seguro que además les resultan familiares.

Existen muchos providers de alto nivel que pueden ser diseñados a medida para proveernos de una documentada abstracción del kernel (Ver la [Guía DTrace](#), ej; proc, io, sched, sysinfo, vminfo), que hacen *tracing* al kernel mucho mejor y más fácilmente de lo que se pueda creer inicialmente.

Comprender el kernel de Solaris es necesario para escribir scripts DTrace avanzados para los que de momento no existe un provider de alto nivel. Por ejemplo para examinar la actividad TCP/IP en detalle. Para ésta cuestión se recomienda leer el libro [Solaris Internals 2nd Edition](#).

#### 1.5.- ¿Qué privilegios necesito para utilizar DTrace?

Sólo ciertos grupos de privilegios permiten funcionar a DTrace. Generalmente el privilegio `dtrace_user` permite acceso a llamadas al sistema (`syscall`) y perfiles de proveedores (`providers`). El privilegio `dtrace_proc` permite añadir acceso al PID del provider y a otros providers basados en USDT. Por ejemplo, sin el acceso al privilegio `dtrace_doc` puedes tener limitada la capacidad de monitorización con agentes DVM dentro de una Máquina Virtual de Java (JVM), pero puedes ver algunas probes de llamadas al sistema.

Si no estás seguro de los privilegios que tienes actualmente, ejecutando el comando `'ppriv $$'` , obtendrás los privilegios que tu shell te ha proporcionado.

### 1.6.- ¿Existe DTrace para otros sistemas operativos?

No, de momento no. DTrace fue inventado por los ingenieros de Sun Microsystems [Bryan Cantrill](#), [Mike Shapiro](#) y [Adam Leventhal](#) para Solaris 10 y OpenSolaris. Hay un proyecto en vía de desarrollo para portar DTrace a FreeBSD, que además ya está a un nivel avanzado, ejecutando algunas de las principales características. A muchos de los ingenieros de DTrace les gustaría verlo en funcionamiento en otros sistemas operativos, de hecho apoyan que DTrace sea portado a otras plataformas.

De todas maneras, Apple ya [anunció](#) que DTrace tendrá mucha influencia en las herramientas de desarrollo para Mac OS X 10.5, más conocido como "Leopard", previsto para Otoño del 2007.

### 1.7.- ¿No se había inventado ya algo similar hace 20 años para los mainframes?

No, DTrace puede probar dinámicamente cualquier función de entrada/retorno en un kernel en funcionamiento (unos 36.000 probes); incluso cualquier función en el código de espacio de usuario y librerías (por ejemplo, mozilla y sus librerías suman unas 100.000 probes); instrucciones a nivel de usuario (sobre los 200.000 probes sólo para la shell Bourne) y sin perder rendimiento.

### 1.8.- ¿Hay libros sobre DTrace?

Sí, existen de momento dos libros excelentes sobre DTrace:

“[The DTrace Guide](#)” es la mejor referencia para DTrace que cubre el lenguaje, providers y montones de ejemplos más. Fue escrito por los ingenieros de DTrace y es una referencia obligada. El libro entero está [online](#) gratuitamente y también se puede bajar en formato PDF.

“[Solaris Performance and Tools](#)” demuestra el uso de DTrace para la observación y el depurado del rendimiento. Fué escrito por Brendan Gregg (autor del DTraceToolkit), Richard McDougall y Jim Mauro (autor de “[Solaris Internals](#)”).

### 1.9.- ¿Existe algún caso de éxito empleando DTrace?

Existen de momento unos cuantos casos de éxito documentados en la red. Realmente muchos de los ingenieros, consultores, NDAs etc. suelen utilizar habitualmente DTrace, pero lamentablemente no pueden mostrar al mundo los detalles de pruebas y monitorización de sus clientes.

De momento, podemos revisar la documentación de Jarod Jenson, el consultor con más experiencia del mundo en DTrace. Jarod ha sido entrevistado por las revistas [SysAdmin Magazine](#) y [ACM](#); en dichas entrevistas podemos ver algunos casos de éxito reales empleando DTrace.

[Brendan Gregg](#) (DTraceToolkit) también ha documentado algunos ejemplos interesantes de análisis con DTrace, realizando la mayor colección de scripts hasta el momento.

## 2.- ¿Cómo funciona DTrace y el lenguaje D?

El lenguaje de programación D está basado en el lenguaje C, así que algún conocimiento inicial de éste lenguaje puede ayudar a su entendimiento. D es bastante más fácil que C ya que sólo hay que aprender un pequeño número de funciones y tipo de variables para ser capaz de escribir poderosos scripts. Además los programas en D también son similares a los programas escritos en awk, lo cual puede resultar de ayuda.

La Figura 1 ilustra el funcionamiento de DTrace, El comando `dtrace(1M)` utiliza una librería llamada `libdtrace(3LIB)` como punto de entrada de varios “providers” dentro del kernel de Solaris, cada uno de ellos nos ofrece una vista lógica de algunos subsistemas del kernel. El binario `dtrace` puede ser utilizado tanto desde línea de comandos como desde shell a través del lenguaje D.

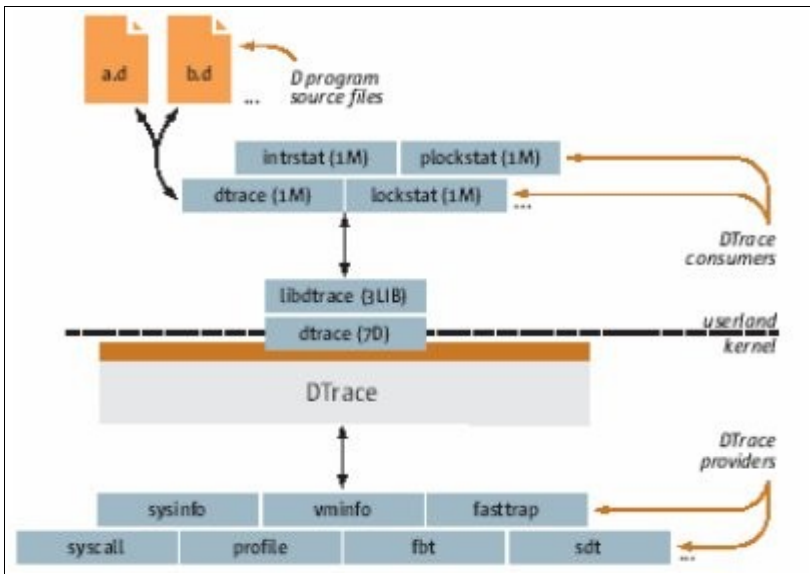


Figura 1: Arquitectura de componentes de DTrace

Cuando se ejecutan, los programas escritos en D son compilados “al vuelo” en bytecodes que pueden ser interpretados dentro del kernel. La máquina virtual de DTrace ejecuta los bytecodes para garantizar que sean seguros. Si el código es seguro y tenemos los suficientes privilegios, el código se parchea dentro del kernel dinámicamente y es ejecutado como código de kernel. Éste es el por qué los probes que no están activos no pueden crear ninguna sobrecarga.

## 2.1.- Qué son los probes y providers?

Una “probe” es un punto de instrumentación que puede ser seguido (tracing) por DTrace. Por ejemplo, llamamos el probe "syscall:read:entry" cuando invocamos la llamada del sistema (syscall) read(2) y se llama a "syscall:read::return" cuando se completa la syscall read(2).

Un ejemplo de probe:      io:nfs:nfs\_bio:start

Ejecutando la instrucción dtrace -l podremos ver la lista de probes.

```
leonidas ~ # dtrace -l | wc -l
48722
```

Existen cuatro componentes para el nombre de la prueba: provider:module:function:name (proveedor:módulo:función:nombre), de manera que posteriormente podamos reconocer fácilmente los probes que puedan ser de nuestro interés.

- El “provider” ó proveedor es una colección de ciertos probes, muy similar a una colección de funciones. Por ejemplo, el provider "syscall" nos provee de probes para la entrada y retorno de todas las llamadas al sistema. Al final de ésta guía podemos encontrar una referencia de los providers.
- Los módulos corresponden a los módulos de kernel de Solaris. En caso de crear nuestros propios probes dentro de las aplicaciones, el módulo puede ser la clase ó el código del módulo en el que definimos el probe. Si el probe corresponde a una ubicación específica, el nombre del módulo es donde se localiza la prueba.
- Las funciones son los nombres de las funciones del código en las que están situados los probes.
- El nombre es el componente final del probe nos da una idea de cuando se ejecuta el probe, como por ejemplo BEGIN ó END

### 3.- Ejemplos trabajando con DTrace

#### 3.1.- Llamadas al sistema (syscalls)

Los syscalls pueden ser fácilmente seguidos utilizando el provider syscall, que nos ofrece un probe para la entrada y retorno de la llamada al sistema. Como un punto en el medio entre en el espacio de usuario y el de kernel, la interfaz syscall refleja perfectamente el comportamiento de la aplicación. Cada syscall está documentada en la sección 2 de las páginas del manual (man).

Aquí algunos ejemplos de una sola línea con DTrace:

Ficheros abiertos por nombre del proceso,

```
leonidas ~ # dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0));
}'
dtrace: description 'syscall::open*:entry ' matched 2 probes
CPU ID          FUNCTION:NAME
 0 49565         open:entry SciTE /usr/lib/iconv/alias
 0 49565         open:entry SciTE /usr/lib/iconv/alias
 0 49565         open:entry SciTE /usr/lib/iconv/alias
 0 49565         open:entry SciTE /usr/lib/iconv/alias
 0 49565         open:entry SciTE /usr/lib/iconv/alias
 0 49565         open:entry nscd /etc/security/prof_attr
 0 49565         open:entry in.routed /dev/kstat
```

Contador de llamadas al sistema por nombre del proceso,

```
leonidas ~ # dtrace -n 'syscall:::entry { @num[execname] = count(); }'
dtrace: description 'syscall:::entry ' matched 230 probes
^C

fmd                1
inetd              1
svc.configd       1
svc.startd        1
automountd        3
gconfd-2          3
ssh-agent         6
screen-4.0.2      7
nmbd              14
xfdesktop         16
ipmon             24
tail              24
nscd              26
```

---

env	42
head	48
tr	58
dirname	60
rm	61
xfce4-session	80
uname	92
dsdm	98
firefox-bin	112
xfce-mcs-manager	176
xfterm4	222
init	292
exo-open	399
xfwm4	400
grep	486
dtrace	515
expr	530
netbeans	643
evince	651
xfce4-panel	806
SciTE	918
sh	1115
Terminal	1415
java	3219
Xorg	3422

Contador de llamadas al sistema con syscall,  
leonidas ~ # dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'  
dtrace: description 'syscall:::entry ' matched 230 probes  
^C

fcntl	1
getpid	1
mmap	1
open	1
schedctl	1
fstat64	2
so_socket	2
stat64	2
close	3
sysconfig	3
sigaction	5
brk	6
xstat	10
lwp_sigmask	11
setcontext	11
nanosleep	13
gtime	14

---

write	15
writew	16
setitimer	19
p_online	32
read	56
lwp_park	66
pollsys	239
ioctl	495

Se puede establecer un valor particular para medir el tiempo transcurrido y el tiempo en CPU de las llamadas al sistema, de forma que nos explique el tiempo de carga y respuesta de la CPU. Por ejemplo, la herramienta `procsystime` de `DTraceToolkit` realiza esta función utilizando los flags `-e` y `-o`

### 3.2.- Entrada/Salida de disco (Disk I/O)

Los eventos de disco se pueden seguir empleando el proveedor `io`, el cual nos ofrece probes para la petición y completación de Entrada/Salida de discos y clientes NFS.

Cada probe nos muestra detalles muy extensos sobre la Entrada/Salida a través de la matriz `args[]`, como está documentado en la guía `DTrace`.

Los siguientes listados nos muestran algunos de estas pruebas:

```
leonidas ~ # dtrace -ln 'io:genunix::'
```

ID	PROVIDER	MODULE	FUNCTION NAME
3769	io	genunix	biodone done
3770	io	genunix	biowait wait-done
3771	io	genunix	biowait wait-start
3780	io	genunix	default_physio start
3781	io	genunix	bdev_strategy start
3782	io	genunix	aphysio start

Hay que tener en cuenta los siguientes puntos a la hora de utilizar el provider io para el seguimiento de la actividad de disco:

- Ésta es la petición actual de Entrada/Salida de disco. Tu aplicación puede tener cargas de Entrada/Salida que pueden ser absorbidas por el cache del sistema de ficheros.
- Las I/O Completions (io:::done) son asíncronas, así que pid y execname podrían no identificar los procesos responsables.
- Las peticiones de escritura en disco (io:::start) suelen ser a menudo asíncronas a los procesos responsables, como el sistemas de ficheros ha cacheado la escritura, ya no existirá más adelante en el medio de almacenamiento.
- Los eventos io no significan necesariamente que las cabezas del disco se estén moviendo. Algunos discos tienen buffers para cachear la actividad de Entrada/Salida, especialmente los arrays de discos.

Algunos ejemplos de I/O en una única línea:

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

```
dtrace: description 'io:::start ' matched 3 probes
```

```
CPU ID          FUNCTION:NAME
 0 3781         bdev_strategy:start 0 sched 512
 0 3781         bdev_strategy:start 0 sched 512
 0 3781         bdev_strategy:start 0 sched 1024
 0 3781         bdev_strategy:start 0 sched 1024
 0 3781         bdev_strategy:start 0 sched 4608
 0 3781         bdev_strategy:start 0 sched 4608
 0 3781         bdev_strategy:start 0 sched 3584
 0 3781         bdev_strategy:start 0 sched 3584
 0 3781         bdev_strategy:start 0 sched 3584
 0 3781         bdev_strategy:start 0 sched 7168
 0 3781         bdev_strategy:start 0 sched 7168
```

Agregación de tamaño de disco,

```
leonidas ~ # dtrace -n 'io:::start { @size[execname] = quantize(args[0]->b_bcount); }'
```

```
dtrace: description 'io:::start ' matched 3 probes
```

```
^C
```

```
sched
```

```
value ----- Distribution ----- count
 256 |                                     0
 512 | @@@@@@@@@@@@@@                       9
1024 | @@@@@@@@@@@@@@@@@@                   13
```

---

2048	@@@@	0	16
4096		0	
8192		0	
16384	@@@@	4	
32768		0	

Otros ejemplos: el famoso “Hola, mundo!”,

Ya en nuestro editor de textos favorito escribimos el siguiente código y lo guardamos con el nombre hola.d :

```
BEGIN
{
    trace("Hola, mundo!");
    exit(0);
}
```

Para ejecutar el código debemos utilizar el comando dtrace con el flag -s :

```
leonidas ~ # dtrace -s hola.d
dtrace: script 'hola.d' matched 1 probe
CPU   ID          FUNCTION:NAME
0     1              :BEGIN  Hola, mundo!
```

Para más información:

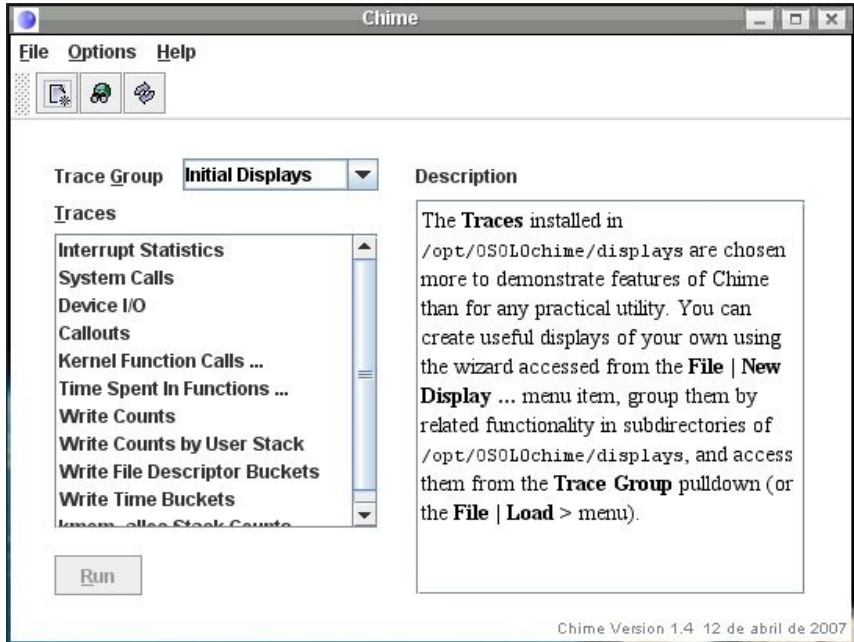
En [DTraceToolkit](#) podremos encontrar numerosos scripts más que nos sirvan de ayuda.

Si tienes instalado Solaris Express Developer Edition, en el directorio /usr/demo/dtrace encontrarás más scripts.

También puedes usar tu navegador web para verlos todos, escribiendo la URL: <file:///usr/demo/dtrace/index.html> en la barra de dirección.

## 4.- CHIME, un interfaz gráfico para DTrace

CHIME es una herramienta gráfica que nos permite revisar visualmente lo que hace DTrace. Es una alternativa a las utilidades de línea de comandos (como por ejemplo intrstat) CHIME es más intuitivo y potencialmente más útil. En particular, la habilidad de visualizar los datos sobre el tiempo añade una dimensión que nos faltaba para la observación del sistema.



Pantalla inicial de CHIME

Una característica importante de CHIME es la habilidad de añadir nuevos displays (vistas) sin tener que recompilar (las vistas se describen en ficheros XML). Su sintaxis nos permite que los valores de cada programa DTrace puedan ser sustituidos detrás de cada escenario, permitiendo al usuario de CHIME seleccionar de una gama descrita las modificaciones del programa sin tener que saber nada sobre DTrace.

CHIME se puede bajar desde la [página del proyecto chime-project](#). Si creas una vista útil para CHIME, por favor compártela en [dtrace-discuss](#) junto con cualquier anécdota relacionada y así podrán agregarla a la página del proyecto para que otros puedan usarla.

Para ejecutar CHIME es necesario tener al menos el build 35 de Solaris Nevada.

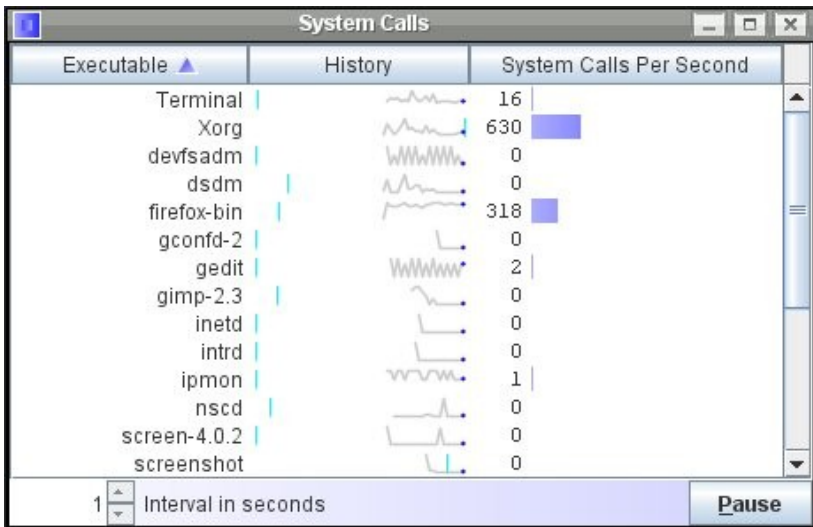
La ubicación de los binarios de CHIME se encuentran en /opt/OSOL0chime/bin/chime

Para usar CHIME debes tener suficientes privilegios DTrace. Ésto no es problema si somos root, sin embargo quizás necesites ejecutar la instrucción xhost +a para que la aplicación Java de CHIME puede funcionar sobre nuestro DISPLAY.

Una mejor forma de obtener permisos DTrace es añadir la siguiente línea a /etc/user\_attr (reemplaza <user> con tu usuario):

```
iban:::defaultpriv=basic,dtrace_proc,dtrace_kernel
```

El privilegio dtrace\_kernel nos da acceso de sólo lectura a la máquina. Necesitamos volver a logearnos en el sistema si queremos que los efectos se apliquen.



Vista de las llamadas al sistema