

# Tema 3. Monitorización de servicios y programas

*¿Cómo medir el rendimiento de mi servidor?*

Analistas, administradores y diseñadores



# Objetivos del tema

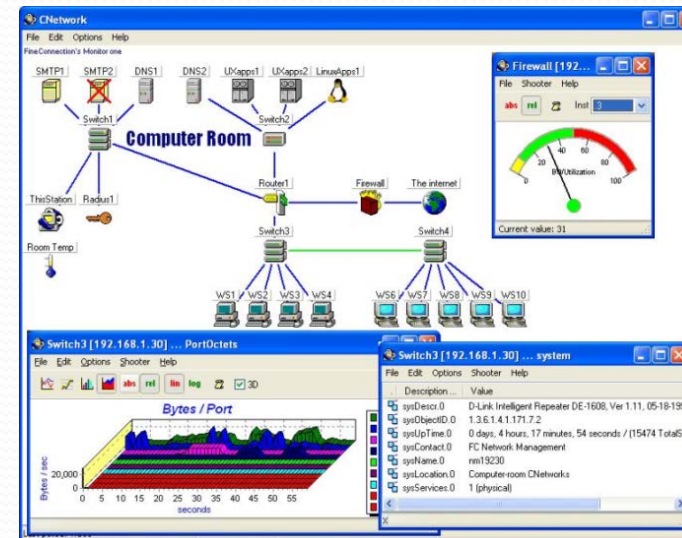
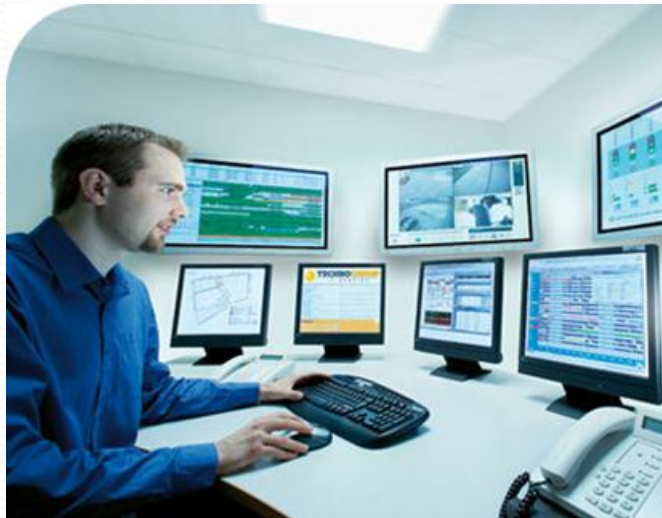
- Entender el concepto de monitor de actividad de un servidor y sus diferentes utilidades e implementaciones.
- Conocer las características fundamentales de un monitor a nivel de sistema operativo y a nivel de aplicación concreta (profilers).
- Comprender el papel que desempeñan los monitores para evaluar el rendimiento de un servidor ante una carga real.
- Saber interpretar adecuadamente la información que aporta un monitor.

# Bibliografía

- *Evaluación y modelado del rendimiento de los sistemas informáticos*. Xavier Molero, C. Juiz, M. Rodeño. Pearson Educación, 2004.
  - Capítulo 2
- *Measuring computer performance: a practitioner's guide*. D. J. Lilja, Cambridge University Press, 2000.
  - Capítulos 4 y 6
- *The art of computer system performance analysis*. R. Jain. John Wiley & Sons, 1991.
  - Capítulos 7 y 8
- *System performance tuning*. G.D. Musumeci, M. Loukides. O'Reilly, 2002.
  - Capítulo 2
- *Linux performance and tuning guidelines*. E.Ciliendo, T.Kunimasa. IBM Redpaper, 2007.
  - Capítulos 1 y 2
- *Linux performance tools*. B. Gregg. 2015. <https://conferences.oreilly.com/velocity/devops-web-performance-2015/public/schedule/detail/42513>.
- *Linux man pages*. <http://www.linuxmanpages.com/>.

# Contenido

- Concepto de monitor de actividad.
- Monitorización a nivel de sistema.
- Monitorización a nivel de aplicación.



## 3.1. Concepto de Monitor de Actividad



# ¿Para qué monitorizar un servidor?

- Administrador/Ingeniero
  - Conocer cómo se usan los recursos para saber:
    - Qué hardware hay que reconfigurar / sustituir/ añadir (cuello de botella).
    - Qué parámetros del sistema hay que ajustar.
  - Obtener un modelo de un componente o de todo el sistema para poder deducir qué pasaría si...
  - Poder predecir cómo va a evolucionar la **carga** con el tiempo (*capacity planning*).
  - Tarifcar a los clientes (*cloud computing*).
- Programador
  - Conocer las partes críticas de una aplicación de cara a su optimización (*hot spots*).
- Sistema Operativo
  - Adaptarse dinámicamente a la **carga**.



# La carga y la actividad de un servidor

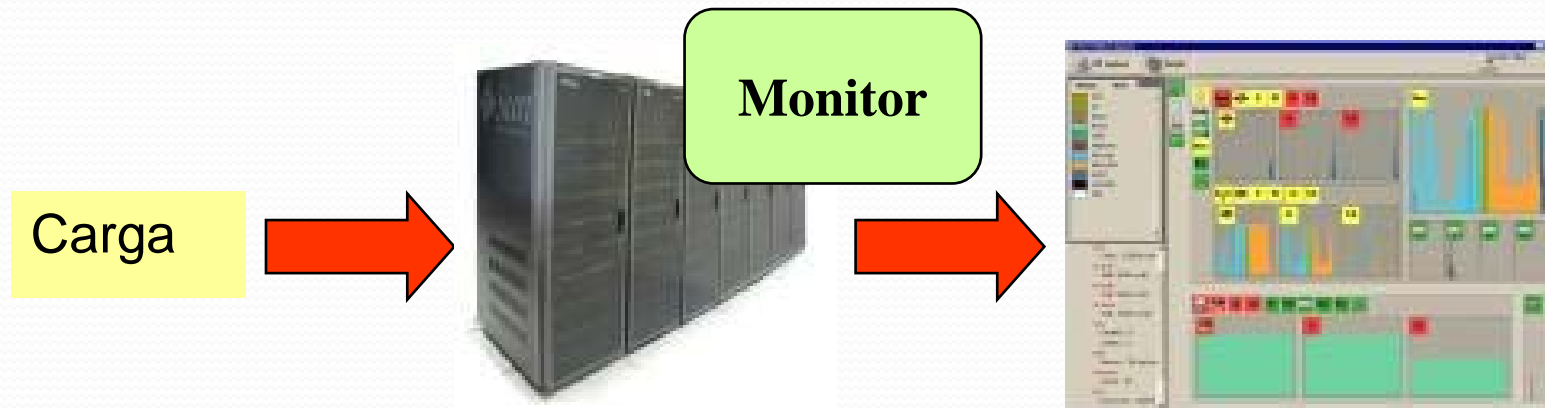
- **Carga** (*workload*): conjunto de tareas que ha de realizar el servidor. (= Todo aquello que demande recursos del servidor.)
- **Actividad** de un servidor: conjunto de operaciones que se realizan en el servidor como consecuencia de la carga que soporta.
- Algunas variables que reflejan la actividad de un servidor:
  - Procesadores: Utilización, temperatura,  $f_{CLK}$ , nº procesos, nº interrupciones, cambios de contexto, etc.
  - Memoria: nº de accesos, memoria utilizada, fallos de caché, fallos de página, uso de memoria de intercambio, latencias, anchos de banda, voltajes, etc.
  - Discos: lecturas/escrituras por unidad de tiempo, longitud de las colas de espera, tiempo de espera medio por acceso, etc.
  - Red: paquetes recibidos/enviados, colisiones por segundo, sockets abiertos, paquetes perdidos, etc.
  - Sistema global: nº de usuarios, nº de peticiones, etc.



Un servidor no es “bueno” ni “malo” *per se*, sino que se adapta mejor o peor a un tipo determinado de carga.

# Definición de monitor de actividad

- Herramienta diseñada para medir la actividad de un sistema informático y facilitar su análisis.



- Acciones típicas de un monitor:
  - Medir alguna/s variables que reflejen la actividad.
  - Procesar y almacenar la información recopilada.
  - Mostrar los resultados.





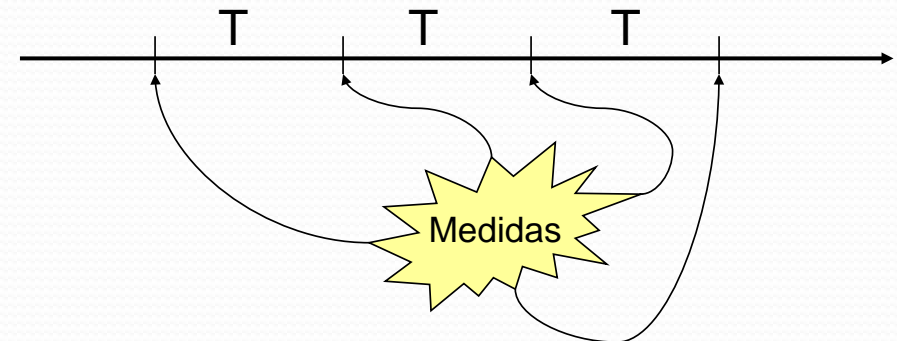
# Tipos de monitores: ¿cuándo se mide?

Cada vez que ocurre un evento (*monitor por eventos*)

- Evento: Cambio en el estado del sistema.
- Mide el nº de ocurrencias de uno o varios eventos.
- Información exacta.
- Ejemplos de eventos:
  - Abrir/cerrar un fichero.
  - Fallo en memoria cache.
  - Interrupción de un dispositivo periférico.

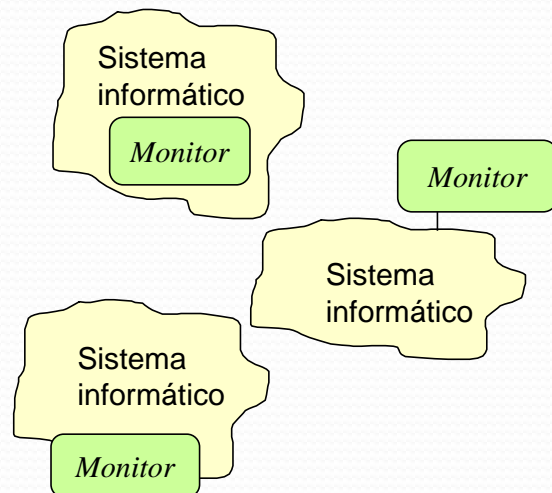
Cada cierto tiempo (*monitor por muestreo*)

- Cada  $T$  segundos, siendo  $T$  el **periodo de muestreo**, el monitor realiza una medida.
- La cantidad de información recogida depende de  $T$ .
- $T$  puede ser, a su vez, variable.
- Información estadística.



# Tipos de monitores: ¿cómo se mide?

- Software:
  - Programas instalados en el sistema
- Hardware
  - Dispositivos físicos de medida (menor sobrecarga)
- Híbridos
  - Utiliza los dos tipos anteriores





```

$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
miguel    29951 55.9   0.1 1448   384 pts/0    R    09:16   0:11 tetris
carlos    29968 50.6   0.1 1448   384 pts/0    R    09:32   0:05 tetris
javier    30023  0.0   0.5 2464  1492 pts/0    R    09:27   0:00 ps aux
    
```



Pin	Name	Color
1	GND	black
2	+12VDC	yellow
3	Sense	green
4	Control	blue



▼ Temperatures				
Mainboard	CPU	CPU1 Core	CPU0 Core	
xx.x°C	xx.x°C	73.0°C	73.0°C	
HDD Temperatures				
 51.0°C	 44.0°C	xx.x°C	xx.x°C	
▼ Cooling fans				
CPU	CPU1	Chassis	Power	
xxxx rpm	xxxx rpm	xxxx rpm	xxxx rpm	
Voltages				
+12V	+xx.xxV	+5V	+x.xxV	Core
-12V	-xx.xxV	-5V	-x.xxV	Aux
▼ Graphics adapters				
Fan speed		Temperatures		Voltages
Fan speed		GPU	Ambient	Core
xxxx rpm	xx%	xx.x°C	xx.x°C	+x.xxV
				Bus
				+x.xxV

# Tipos de monitores: ¿existe interacción con el analista/administrador?

- No existe. La consulta sobre los resultados se realiza aparte mediante otra herramienta independiente al proceso de monitorización: monitores tipo batch, por lotes o en segundo plano (*batch monitors*).
- Sí existe. Durante el propio proceso de monitorización se pueden consultar los valores monitorizados y/o interactuar con ellos realizando representaciones gráficas diversas, modificando parámetros del propio monitor, etc.: monitores en primer plano o interactivos (*on-line monitors*).



# Atributos más importantes que caracterizan a un monitor de actividad

- **Anchura de Entrada** (*Input Width*): ¿Cuánta información (p.ej. nº de bytes) se almacena por cada medida que toma el monitor?
- **Sobrecarga** (*Overhead*): ¿Qué recursos le “roba” el monitor al sistema? *El instrumento de medida puede perturbar el funcionamiento del sistema.*

$$Sobrecarga_{Recurso}(\%) = \frac{\text{Uso del recurso por parte del monitor}}{\text{Capacidad total del recurso}} \times 100$$

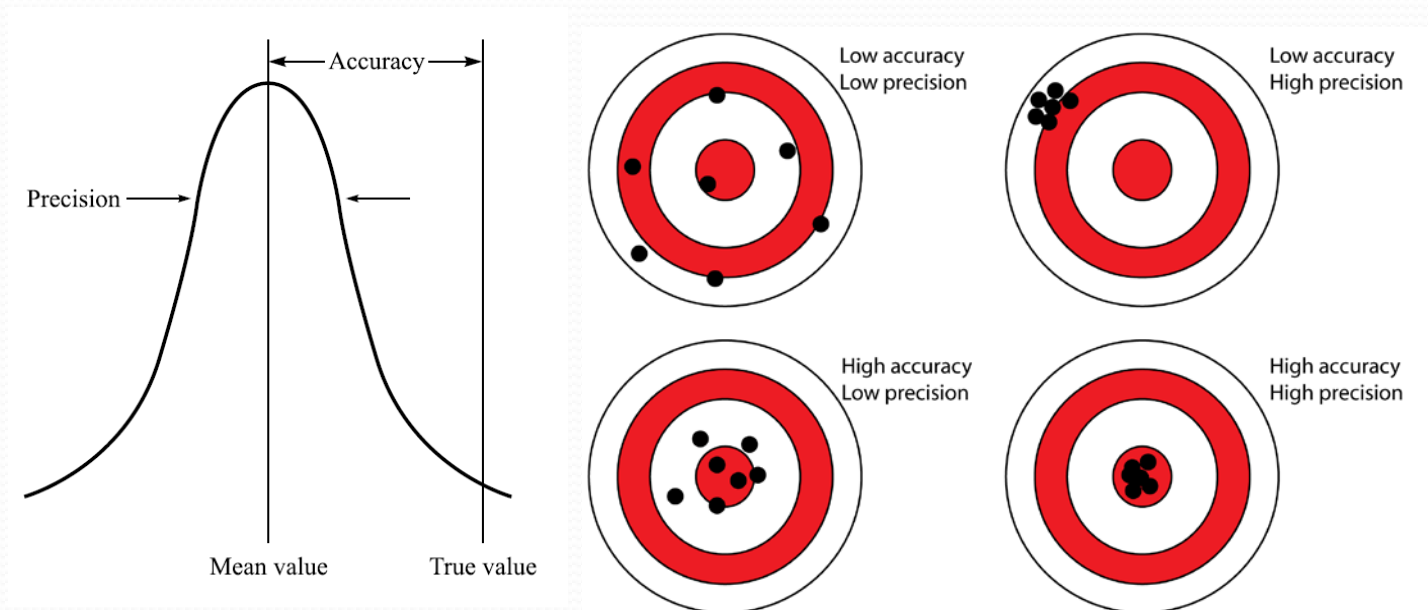
- Ejemplo: Sobrecarga de CPU de un monitor software por muestreo. La ejecución de las instrucciones del monitor se lleva a cabo utilizando recursos del sistema monitorizado. Supongamos que el monitor se activa cada 5s y que cada activación del mismo usa el procesador durante 6 ms.

$$Sobrecarga_{CPU}(\%) = \frac{6 \times 10^{-3}s}{5s} \times 100 = 0,12\%$$



## Otros atributos importantes que caracterizan, en general, a cualquier herramienta de medida (=sensor)

- **Exactitud** del sensor (*Accuracy, offset*): ¿Cómo se aleja el valor medido del valor real que se quiere medir?
- **Precisión** del sensor: Cuando se mide varias veces el mismo valor real, ¿se mide siempre lo mismo?, ¿cuál es la dispersión de las medidas?
- **Resolución** del sensor: ¿Cuánto tiene que cambiar el valor a medir para detectar un cambio?



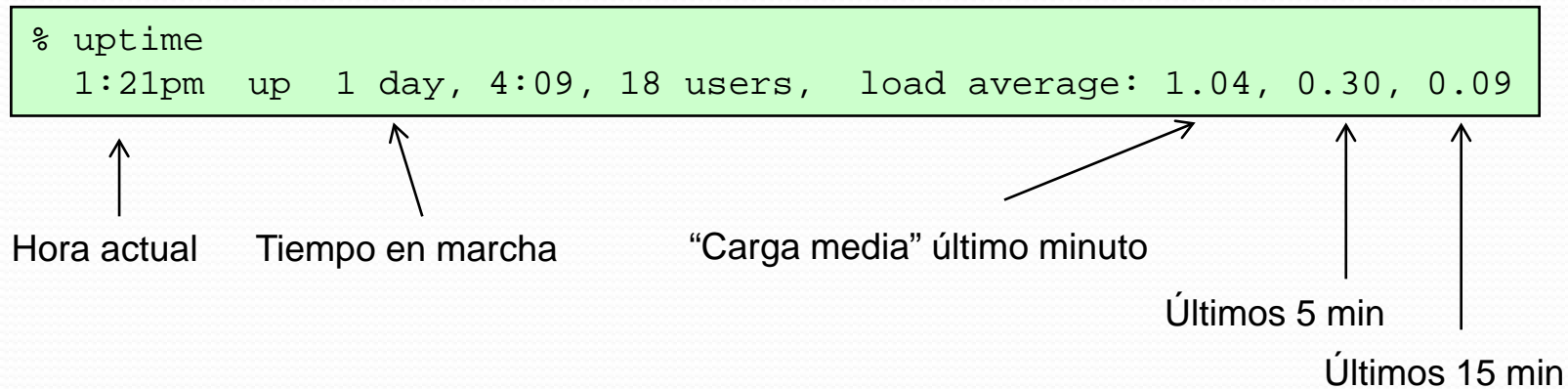
## 3.2. Monitorización a nivel de sistema

# El directorio `/proc` (Linux)

- Es un directorio **virtual** utilizado por el núcleo de Linux para facilitar el acceso del usuario a las estructuras de datos del S.O.
- A través de [`/proc`](#) podemos:
  - Acceder a información global sobre el S.O.: `loadavg`, `uptime`, `cpuinfo`, `meminfo`, `mounts`, `net`, `kmsg`, `cmdline`, `slabinfo`, `filesystems`, `diskstats`, `devices`, `interrupts`, `stat`, `swap`, `version`, `vmstat`, ...
  - Acceder a la información de cada uno de los procesos del sistema (`/proc/[pid]`): `stat`, `status`, `statm`, `mem`, `smaps`, `cmdline`, `cwd`, `environ`, `exe`, `fd`, `task`...
  - Acceder y, a veces, modificar algunos parámetros del kernel del S.O. (`/proc/sys`): `dentry_state`, `dir-notify-enable`, `dquot-max`, `dquot-nr`, `file-max`, `file-nr`, `inode-max`, `inode-nr`, `lease-break-time`, `mqueue`, `super-max`, `super-nr`, `acct`, `domainname`, `hostname`, `panic`, `pid_max`, `version`, `net`, `vm`...
- En Linux, la mayoría de los monitores de actividad a nivel de sistema usan como fuente de información este directorio.

# uptime

- Tiempo que lleva el sistema en marcha y la “carga media” que soporta.



[man uptime \(http://man7.org/linux/man-pages/man1/uptime.1.html\)](http://man7.org/linux/man-pages/man1/uptime.1.html)

## NAME

uptime - Tell how long the system has been running.

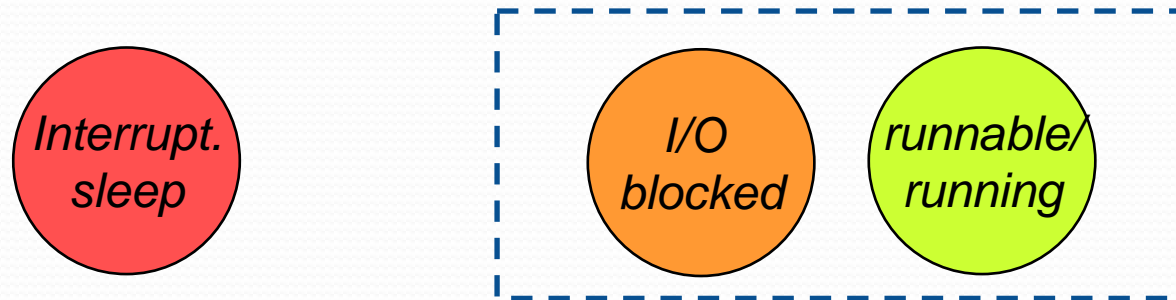
## DESCRIPTION

- **uptime** gives a one line display of the following information. The current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes. This is the same information contained in the header line displayed by `w`.



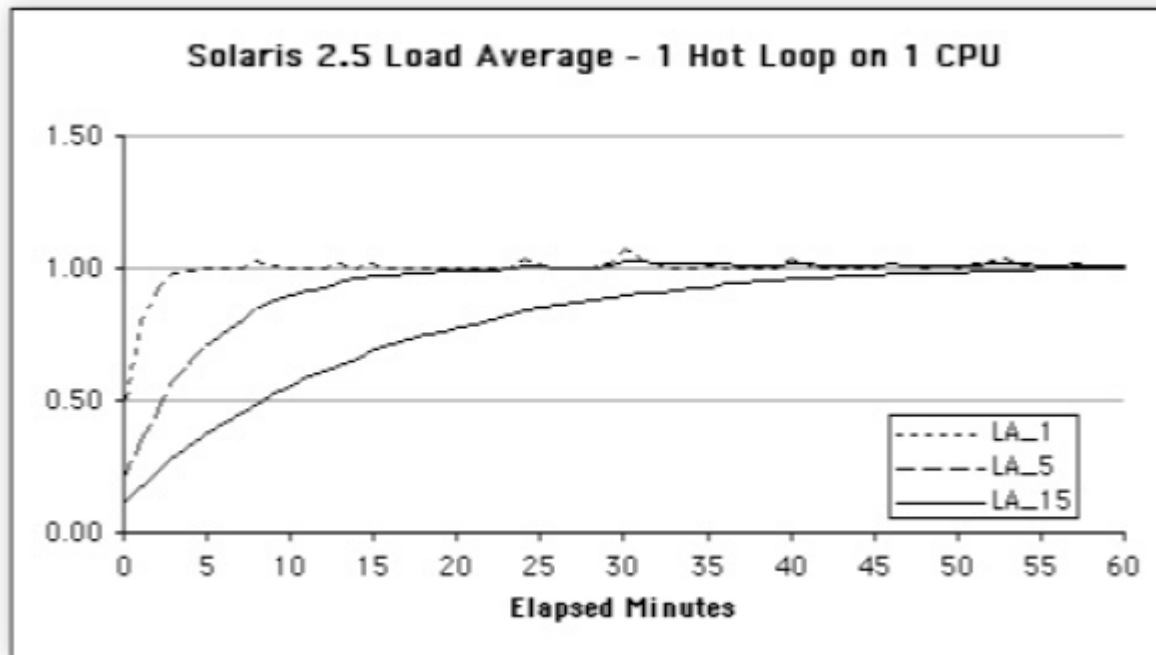
# “Carga” del sistema según Linux

- Estados básicos de un proceso:
  - En ejecución (*running*) o esperando que haya un núcleo (*core*) libre para poder ser ejecutado (*runnable*). La cola de procesos (*run queue*) está formada por aquellos que se están ejecutando y los que pueden ejecutarse (*runnable* + *running*).
  - Bloqueado esperando a que se complete una operación de E/S para continuar (*uninterruptible sleep* = *I/O blocked*).
  - Durmiendo esperando a un evento del usuario o similar (p.ej. una pulsación de tecla) (*interruptible sleep*).
- “Carga del sistema” (*system load*): número de procesos en modo *running*, *runnable* o *I/O blocked*.



# ¿Cómo mide la carga media el S.O.?

- Experimento: Ejecutamos 1 único proceso (bucle infinito). Llamamos a *uptime* cada cierto tiempo y representamos los resultados.



Según *sched.c*, *sched.h* (kernel de Linux):

$$LA(t) = c \cdot load(t) + (1-c) \cdot LA(t-5)$$

- LA (t) = Load Average en el instante t.
- Se actualiza cada 5 segundos su valor.
- load(t) es la “carga del sistema” en el instante t.
- c es una constante. A mayor valor, más influencia tiene la carga actual en el valor medio ( $c_1 > c_5 > c_{15}$ ). Si  $c = c_1$  calculamos LA\_1(t), etc.

# ps (*process status*)

- Información sobre el estado actual de los procesos del sistema.

```
$ ps aur
USER      PID %CPU %MEM  VSZ   RSS TTY      STAT START TIME COMMAND
miguel    29951 55.9   0.1 1448   384 pts/0    R    09:16 0:11 tetris
carlos    29968 50.6   0.1 1448   384 pts/0    R    09:32 0:05 tetris
javier    30023  0.0   0.5 2464  1492 pts/0    R    09:27 0:00 ps aur
```

- USER: Usuario que lanzó el proceso.
- %CPU, %MEM: Porcentaje de procesador y memoria física usada.
- RSS (*resident set size*): Memoria (KiB) física ocupada por el proceso.
- STAT. Estado en el que se encuentra el proceso:
  - **R** (*running or runnable*), **D** (*I/O blocked*),
  - **S** (*interruptible sleep*), **T** (*stopped*),
  - **Z** (*zombie: terminated but not died*).
  - **N** (*lower priority = running niced*),
  - **<** (*higher priority = not nice*).
  - **s** (*session leader*),
  - **+** (*in the foreground process group*),
  - **W** (*swapped/paging*).

## Procesos a mostrar:

-A, -e: show all processes; T: all processes on this terminal; U: processes for specified users...

**Campos que mostrar:** process ID, cumulative user time, number of minor/major page faults, parent process ID, RSS, time process was started, user ID number, user name, total VM size in bytes, kernel scheduling priority, etc.

```
% strace -e open ps
```

```
...
open("/proc/1/stat", O_RDONLY)      = 6
open("/proc/1/status", O_RDONLY)    = 6
open("/proc/1/cmdline", O_RDONLY)   = 6
...
```

# top

- Muestra cada T segundos: carga media, procesos, consumo de memoria...
- Normalmente se ejecuta en modo interactivo (se puede cambiar T, las columnas seleccionadas, la forma de ordenar las filas, etc.)

```
8:48am up 70 days, 21:36, 1 user, load average: 0.28, 0.06, 0.02
Tareas: 47 total, 2 running, 45 sleeping, 0 stopped, 0 zombie
%Cpu(s): 99.6 us, 0.3 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 256464 total, 234008 used, 22456 free, 13784 buffers
KiB Swap: 136512 total, 4356 used, 132156 free, 5240 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	STAT	%CPU	%MEM	TIME	COMMAND
9826	carlos	0	0	388	388	308	R	99.6	0.1	0:22	simulador
9831	miguel	19	0	976	976	776	R	0.3	0.3	0:00	top
1	root	20	0	76	64	44	S	0.0	0.0	0:03	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00	keventd
4	root	20	19	0	0	0	SN	0.0	0.0	0:00	ksoftiq
5	root	20	0	0	0	0	S	0.0	0.0	0:13	kswapd
6	root	2	0	0	0	0	S	0.0	0.0	0:00	bdfush
7	root	20	0	0	0	0	S	0.0	0.0	0:10	kdated
8	root	20	0	0	0	0	S	0.0	0.0	0:01	kinoded
11	root	0	-20	0	0	0	S<	0.0	0.0	0:00	recoved

wa: %tiempo ocioso esperando E/S. st: %tiempo robado por el hipervisor.



# vmstat (*virtual memory statistics*)

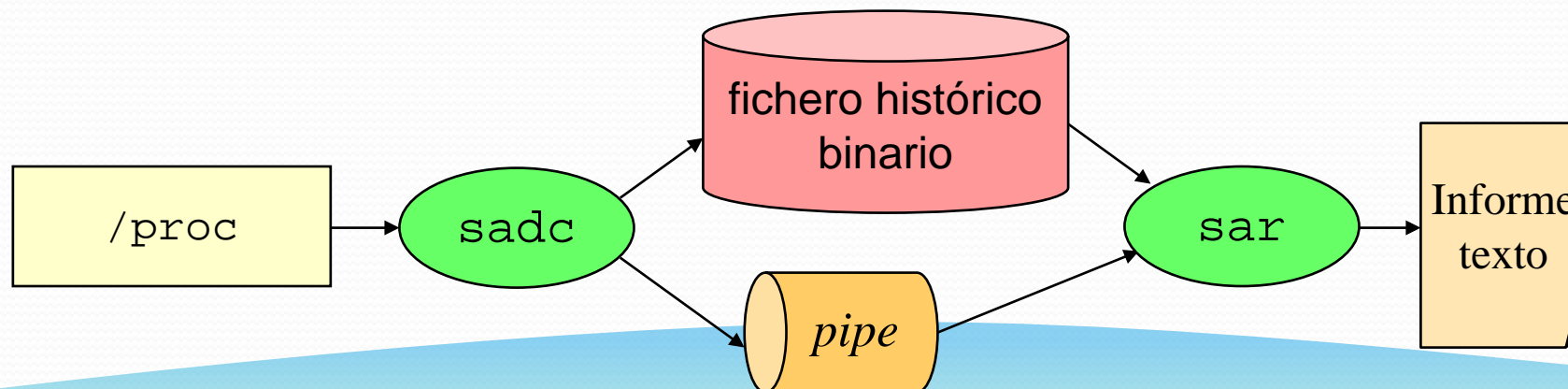
- *Paging* (paginación), *swapping*, interrupciones, cpu
  - La primera línea no sirve para nada (info desde el inicio del sistema)

% vmstat 1 6																
procs		memory				swap		io		system		cpu				
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st
0	0	868	8964	60140	342748	0	0	23	7	222	199	1	4	80	15	0
0	0	868	8964	60140	342748	0	0	0	14	283	278	0	7	80	23	0
0	0	868	8964	60140	342748	0	0	0	0	218	212	6	2	93	0	0
0	0	868	8964	60140	342748	0	0	0	0	175	166	3	3	94	0	0
0	0	868	8964	60140	342752	0	0	0	2	182	196	0	7	88	5	0
0	0	868	8968	60140	342748	0	0	0	18	168	175	3	8	69	20	0

- Procesos: *r* (*running o runnable*), *b* (*I/O blocked*)
- Bloques por segundo transmitidos: *bi* (*blocks in*), (*blocks out*)
- KB/s entre memoria y disco: *si* (*swapped in*), *so* (*swapped out*)
- *in* (*interrupts per second*), *cs* (*context switches per second*)
- Con otros argumentos, puede dar información sobre acceso a discos (en concreto la partición de swap) y otras estadísticas de memoria.

# El monitor `sar` (system activity reporter)

- Forma parte del paquete de monitorización de la actividad Sysstat, mantenido por Sebastien Godard: <http://sebastien.godard.pagesperso-orange.fr/>
- Recopila información sobre la actividad del sistema.
  - Actual: qué está pasando ahora mismo en el sistema (modo interactivo).
  - Histórica: qué ha pasado en el sistema (modo **no** interactivo).
    - Ficheros históricos en `/var/log/sysstat/saDD`, donde los dígitos DD indican el día del mes.
- Esquema de funcionamiento:
  - `sadc` (*system-accounting data collector*): Recoge los datos estadísticos (lectura de contadores) y construye un registro en formato binario (*back-end*).
  - `sar`: Lee los datos binarios que recoge `sadc` y los traduce a texto plano (*front-end*).



# Parámetros de sar

- Gran cantidad de parámetros (puede funcionar en modo batch o en modo interactivo)

**Modo interactivo:** [tiempo\_muestreo, [nº muestras]]

**Modo no interactivo:**

**-f** Fichero de donde extraer la información, **por defecto: hoy**  
**-s** Hora de comienzo de la monitorización  
**-e** Hora de fin de la monitorización

**-u** Utilización global del procesador (**opción por defecto**)  
**-P** Mostrar estadísticas por cada núcleo (-P ALL: todos)  
**-I** Estadísticas sobre interrupciones  
**-w** Cambios de contexto  
**-q** Tamaño de la cola y carga media del sistema  
**-b** Estadísticas globales de transferencias de E/S  
**-d** Transferencias para cada disco  
**-n** Conexión de red  
**-r** Utilización de memoria  
**-R** Estadísticas sobre la memoria  
**-A** Toda la información disponible  
...

# Ejemplo de uso del monitor sar

- Utilización global del procesador (que puede ser multi-núcleo) recopilada durante el día de hoy:

```
$ sar (=sar -u)
00:00:00 CPU %usr %nice %sys %wa %st %idle
00:05:00 all 0.09 0.00 0.08 0.00 0.00 99.83
00:10:00 all 0.01 0.00 0.01 0.00 0.00 99.98
00:15:00 ...
```

- Utilización desglosada de cada núcleo (core) recopilada de forma interactiva cada 1s:

```
$ sar -P ALL 1
19:30:39 CPU %usr %nice %sys %wa %st %idle
19:30:40 all 53.45 0.00 6.18 0.00 0.00 40.37
19:30:40 0 49.49 0.00 5.05 0.00 0.00 45.45
19:30:40 1 51.61 0.00 6.45 0.00 0.00 41.94
19:30:40 2 58.16 0.00 8.17 0.00 0.00 33.67
19:30:40 3 54.55 0.00 5.05 0.00 0.00 40.40
19:30:40 CPU %usr %nice %sys %wa %st %idle
19:30:41 all 50.49 0.00 6.19 0.00 0.00 43.32
...
```



# Monitorización de las unidades de almacenamiento con sar

- Estadísticas globales del sistema de E/S (sin incluir la red) recopiladas entre las 10 y las 12h del día 6 de este mes :

```
$ sar -b -s 10:00:00 -e 12:00:00 -f /var/log/sysstat/sa06
10:00:00      tps      rtps      wtps      bread/s      bwrtn/s
10:05:00      0.74      0.39      0.35         7.96         3.27
10:10:00     65.12     59.96      5.16     631.62     162.64
10:15:00      ...
```

- Información de prestaciones de cada disco recopilada de forma interactiva cada 10s (2 muestras):

```
$ sar -d 10 2
18:46:09 DEV  tps    rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz  await  svctm  %util
18:46:19 sda   1.70     33.60     0.00     19.76     0.00     0.47   0.47   0.08
18:46:19 sr0   0.00      0.00     0.00      0.00     0.00     0.00   0.00   0.00
18:46:19 DEV  tps    rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz  await  svctm  %util
18:46:29 sda   8.60    114.40    518.10    73.55     0.06     7.12   0.93   0.80
18:46:29 sr0   0.00      0.00     0.00      0.00     0.00     0.00   0.00   0.00
```

# Monitorización de la red con sar

- Se puede particularizar la monitorización a una interfaz de red concreta, a un protocolo concreto, se pueden mostrar solo información de errores, etc.
- Ejemplo: Mostramos información recopilada de forma interactiva cada 1s sobre todo el tráfico TCP, incluyendo errores en los paquetes, de todos los dispositivos de red:

```
$ sar -n TCP,ETCP,DEV 1
Linux 3.2.55 (test-e4f1a80b)      08/18/2014      _x86_64_ (8 CPU)

09:10:43 PM  IFACE  rxpck/s  txpck/s  rxkB/s  txkB/s  rxcmp/s  txcmp/s  rxmcsst/s
09:10:44 PM      lo    14.00   14.00    1.34    1.34    0.00    0.00    0.00
09:10:44 PM   eth0 4114.00 4186.00 4537.46 28513.24 0.00    0.00    0.00

09:10:43 PM  active/s  passive/s      iseg/s      oseg/s
09:10:44 PM    21.00      4.00    4107.00   22511.00

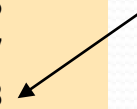
09:10:43 PM  atmptf/s  estres/s  retrans/s  isegerr/s  orsts/s
09:10:44 PM    0.00    0.00    36.00    0.00    1.00
[...]
```

# Almacenamiento de los datos muestreados por sadc

- Se utiliza un fichero histórico de datos por cada día.
- Se programa la ejecución de sadc un número de veces al día con la utilidad “cron” de Linux.
  - Por ejemplo, una vez cada 5 minutos comenzando a las 0:00 de cada día.
- Cada ejecución de sadc añade un registro binario con los datos recogidos al fichero histórico del día.

```
%ls /var/log/sysstat
-rw-r--r-- 1 root root 3049952 Oct 1 23:55 sa01
-rw-r--r-- 1 root root 3049952 Oct 2 23:55 sa02
-rw-r--r-- 1 root root 3049952 Oct 3 23:55 sa03
-rw-r--r-- 1 root root 3049952 Oct 4 23:55 sa04
-rw-r--r-- 1 root root 3049952 Oct 5 23:55 sa05
-rw-r--r-- 1 root root 3049952 Oct 6 23:55 sa06
-rw-r--r-- 1 root root 3049952 Oct 7 23:55 sa07
-rw-r--r-- 1 root root 2372320 Oct 8 18:45 sa08
```

Día actual



# Cálculo de la anchura de entrada del monitor

- Datos de partida:

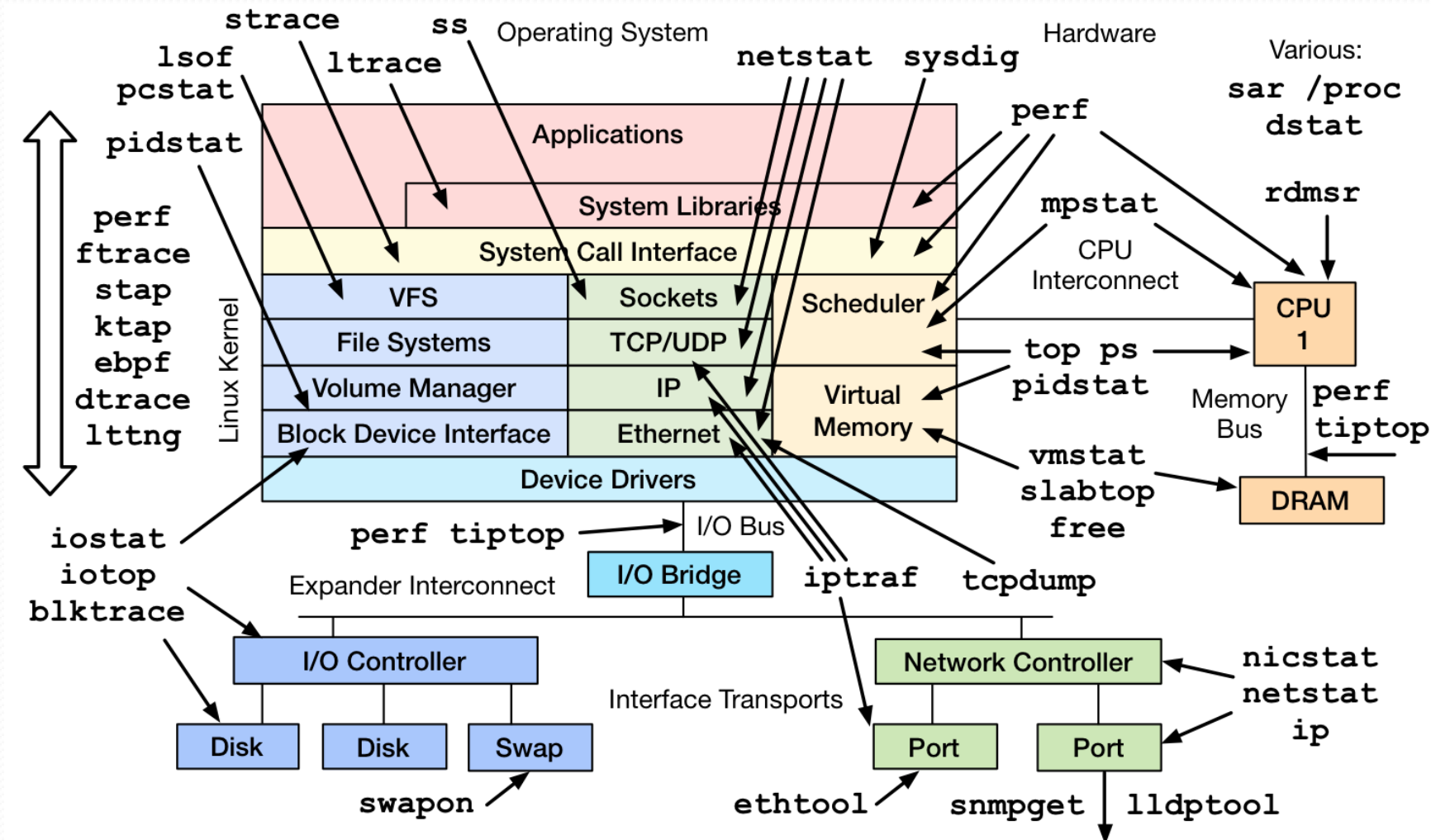
- Extracto de ``ls /var/log/sysstat``:

<code>-rw-r--r--</code>	<code>1</code>	<code>root</code>	<code>root</code>	<code>3049952</code>	<code>Oct 2</code>	<code>23:55</code>	<code>sa02</code>
-------------------------	----------------	-------------------	-------------------	----------------------	--------------------	--------------------	-------------------

Fichero sa02  
(día 2 de octubre)

- Suponemos que la primera muestra se toma a las 0:00 de cada día y que sadc se ejecuta con un tiempo de muestreo constante.
- Solución:
  - El fichero histórico de un día ocupa 3.049.952 bytes (aproximadamente 3,05 MB o 2,91MiB).
  - La orden sadc se ejecuta cada 5 minutos.
    - Cada hora se recogen 12 muestras.
    - Al día se recogen  $24 \times 12 = 288$  muestras.
  - Anchura de entrada del monitor:  
Cada registro ocupa, de media, 10590,1 bytes (aproximadamente 10,59 KB o 10,34KiB).

# Otras herramientas para monitorización a nivel de sistema





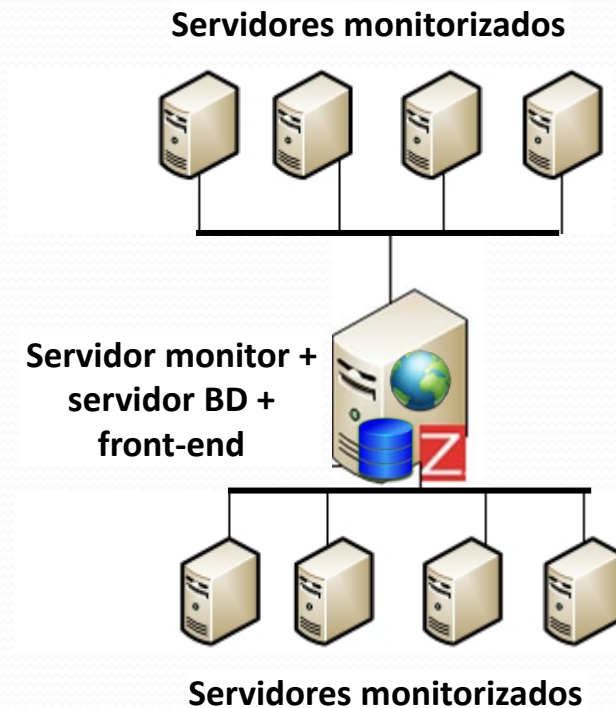
# Procedimiento sistemático de monitorización: método USE (Utilization, Saturation, Errors)

Para cada recurso del servidor (CPU, memoria, almacenamiento, red) comprobamos, para un intervalo de tiempo dado:

- Utilización media: Fracción de uso del recurso (tiempo de CPU, tamaño de memoria, ancho de banda de cada disco, tarjeta de red o módulo de memoria DRAM, etc.)
  - Ejemplo: si el recurso es un determinado núcleo del procesador: fracción de ese intervalo de tiempo en la que el núcleo ha estado ocupado ejecutando instrucciones. Se puede medir con *sar -P* y sumo todos los campos del núcleo que me interesa excepto *%idle* y *%wa*.
- Saturación: Ocupación media de las colas de aquellas tareas que quieren hacer uso de ese recurso.
  - Ejemplo: para un determinado disco duro, ejecuto *sar -d* y leo *avgqu-sz*.
- Errores: Mensajes de error sobre el uso de dichos recursos.
  - Ejemplo: Para DRAM, con *dmesg* puedo ver errores físicos.
- Listado completo: <http://www.brendangregg.com/USEmethod/use-linux.html>

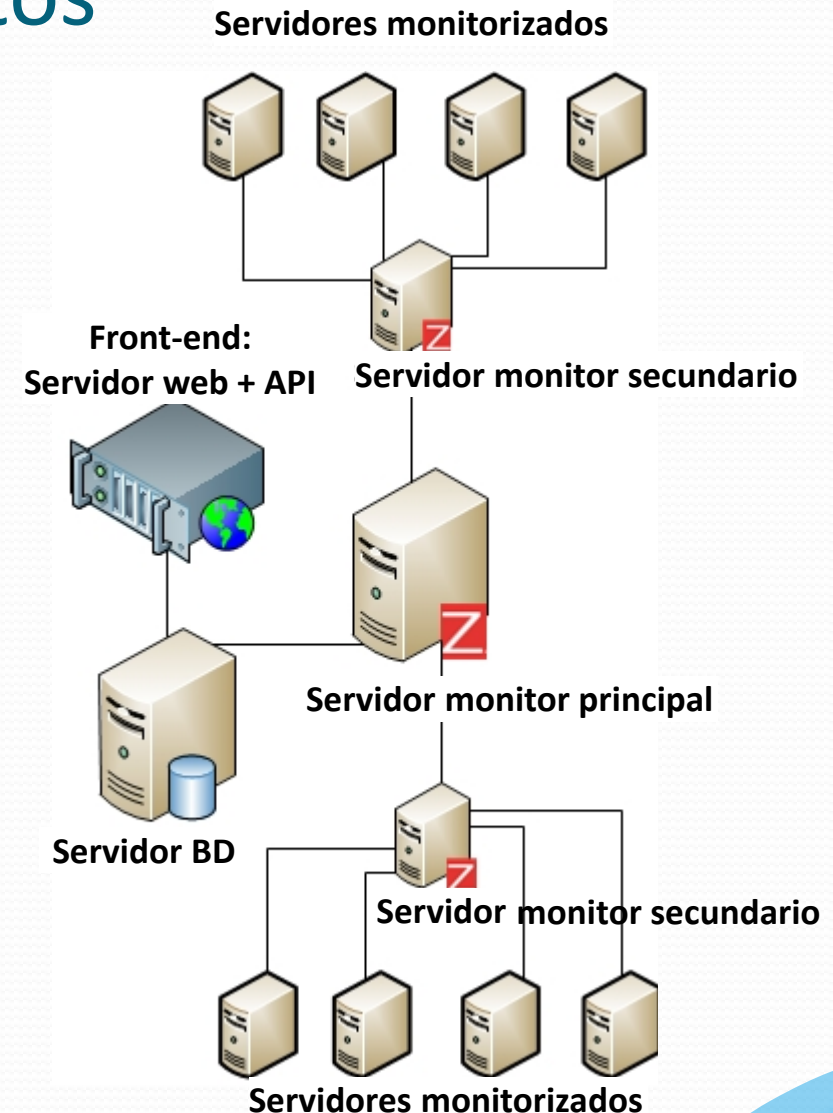
# Herramientas de monitorización de servidores distribuidos

- Cuando tenemos que monitorizar varios servidores de forma simultánea, es más conveniente utilizar herramientas especiales de monitorización de equipos distribuidos en red como Nagios, Naemon, Shinken, Ganglia, Munin, Zabbix o Elastic.



# Algunas características deseables de estos monitores de servidores distribuidos

- Compatibilidad con múltiples protocolos para enviar el valor medido: SNMP, IPMI, HTTP, JMX. Servidores independientes para la base de datos y el front-end. API para aplicaciones secundarias (p.ej. Grafana).
- Escalabilidad: servidores monitores secundarios encargados de subconjuntos de servidores. Auto-descubrimiento de nuevos servidores.
- Extensibilidad: permitir plugins para ampliar la funcionalidad (p.ej. añadir nuevas medidas por medio de scripts, añadir nuevos protocolos, etc.)
- Prestaciones: Permitir tanto *polling* como *pushing*, con agente o sin él. Envío por lotes. Compresión.
- Fiabilidad: Validación de los valores monitorizados.
- Disponibilidad: Gestión de alarmas y envío de notificaciones (email, Telegram, etc.) si se detectan elementos caídos.
- Seguridad: Encriptación de las comunicaciones, tablas hash para evitar suplantación de servidores.



### **3.3. Monitorización a nivel de aplicación (profilers)**

# Monitorización a nivel de aplicación (*profiling*)

- Objetivo de un *profiler*:
  - Monitorizar la actividad generada por una aplicación concreta con el fin de obtener **información para poder optimizar su código**.
- Información que sería interesante que nos proporcionara un *profiler*:
  - ¿Cuánto tiempo tarda en ejecutarse el programa? ¿Qué parte de ese tiempo es de usuario y cuál de sistema? ¿Cuánto tiempo se pierde por las operaciones de E/S?
  - ¿En qué parte del código pasa la mayor parte de su tiempo de ejecución (=hot spots)?
  - ¿Cuánto tiempo tarda en ejecutarse cada función (procedimiento o subrutina) a la que llama el programa?
  - ¿Cuántas veces se llama a cada función del programa y desde dónde?
  - ¿Cuántos fallos de caché/página genera cada línea del programa?
  - ¿Qué cantidad de memoria está ocupada en cada momento del programa? Etc.



# Una primera aproximación: /usr/bin/time

El programa /usr/bin/time mide el tiempo de ejecución de un programa y muestra algunas estadísticas sobre su ejecución.

- User time: tiempo de CPU ejecutando en modo usuario.
- System time: tiempo de CPU ejecutando código del núcleo del S.O.
- Elapsed (wall clock) time: tiempo que tarda el programa en ejecutarse.
- *Major page faults*: fallos de página que requieren acceder al almacenamiento permanente.
- Cambios de contexto voluntarios: Cuando acaba el programa o al tener que esperar a una operación de E/S cede la CPU a otro proceso.
- Cambios involuntarios: Expira su “*time slice*”.

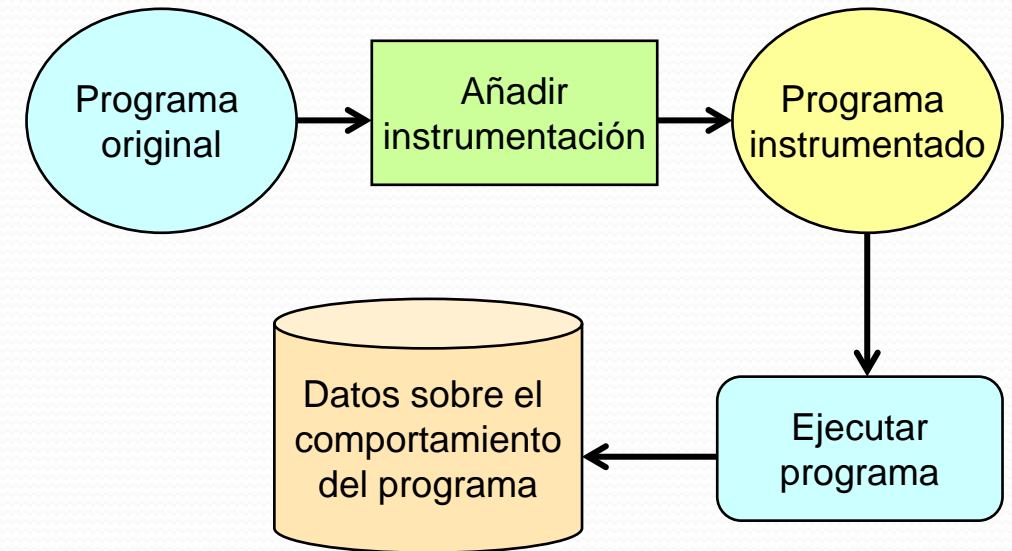
```
% /usr/bin/time -v ./matr_multiplication
User time (seconds): 10.47
System time (seconds): 0.01
Percent of CPU this job got: 99%
Elapsed (wall clock) time 0:10.49
Maximum RSS (kbytes): 12688
Major page faults: 0
Minor page faults: 2982
Voluntary context switches: 1
Involuntary context switches: 924
...
```

## No confundir con:

```
% time ./matr_mult2
real    0m4.862s
user    0m4.841s
sys     0m0.010s
```

# Profiler gprof

- **Estima el tiempo de CPU** que consume cada función de un proceso/hilo. También calcula el número de veces que se ejecuta cada función y cuántas veces una función llama a otra.
- Utilización de gprof para programas escritos en C, C++:
  - Instrumentación en la compilación:
    - `gcc prog.c -o prog -pg -g`
  - Ejecución del programa y recogida de información:
    - `./prog`
    - La información recogida se guarda en el fichero `gmon.out`
  - Visualización de la información referida a la ejecución del programa:
    - `gprof prog`



# ¿Cómo funciona un programa instrumentado por gprof?

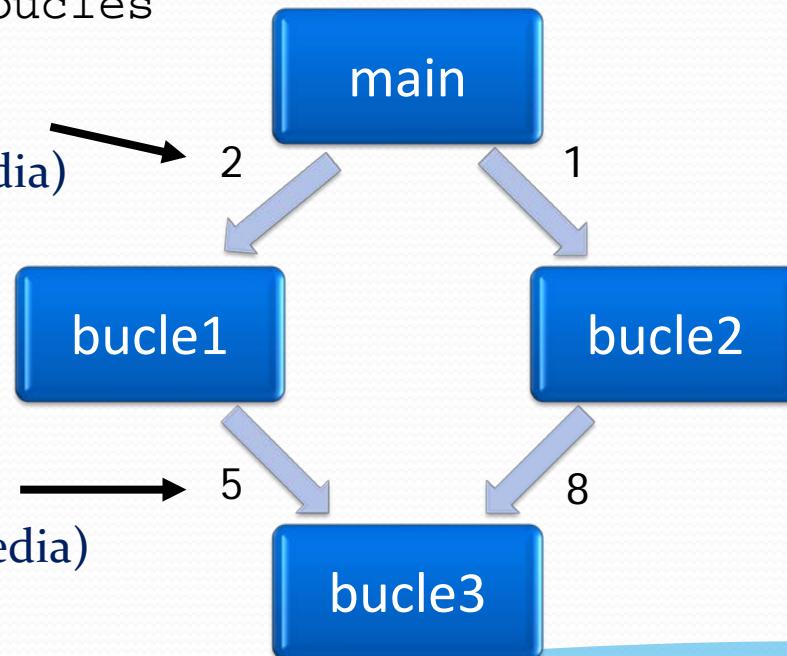
- Arranque del programa:
  - Genera una tabla con la dirección física en memoria de cada función del programa.
  - Se inicializan contadores de **cada función** del programa a 0. Hay dos contadores por función: *c1* para medir el número de veces que se ejecuta y *c2* para estimar su tiempo de CPU.
  - El S.O. programa un temporizador (por defecto 0,01s) que irá decrementándose cada vez que se ejecute código del programa.
- Durante la ejecución del programa:
  - Cada vez que se ejecuta una función se incrementa el contador *c1* asociado a la función. De paso, se mira a través de la pila qué función la ha llamado y se guarda esa información.
  - Cada vez que el temporizador llega a 0s, se interrumpe el programa y se incrementa el contador *c2* de la función interrumpida. Se re-inicia el temporizador.
- Al terminar el programa:
  - Teniendo en cuenta el tiempo total de CPU del programa y los contadores *c2*, se **estima** el tiempo de CPU de cada función.
  - Se generan el *flat profile* y el *call profile* a partir de la información recopilada.

# Ejemplo

- Instrumentación (-pg) en la compilación.
  - gcc bucles.c -pg -g -o bucles
- Ejecución del programa monitorizado.
  - ./bucles
- Obtención de la información recopilada.
  - gprof bucles

Cada ejecución de  
main llama (de media)  
2 veces a bucle1

Cada ejecución de  
bucle1 llama (de media)  
5 veces a bucle3

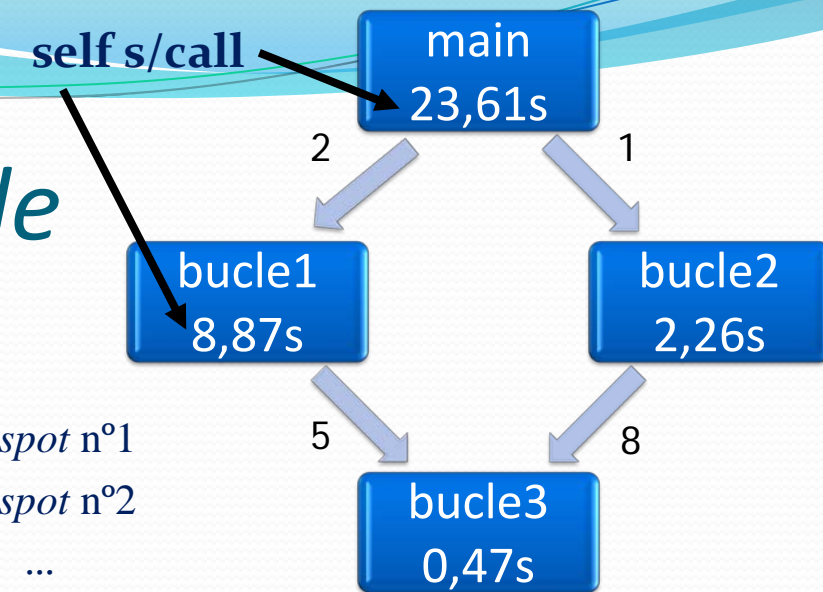


## bucles.c

```
float a=0.3; float b=0.8; float c=0.1;
void main(void) {
    unsigned long i;
    for (i=0;i<160000000;i++) a=a*b/(1+c);
    bucle1(); bucle1(); bucle2();
}
void bucle1(void) {
    unsigned long i;
    for (i=0;i<40000000;i++) {
        c=(c+c*c)/(1+a*c);c=a*b*c; }
    for (i=1;i<=5;i++) bucle3();
}
void bucle2(void) {
    unsigned long i;
    for (i=0;i<100000000;i++) {
        c=(c+c*c+c*c*c)/(1+a*c*c);
        c=a*b*c; }
    for (i=1;i<=8;i++) bucle3();
}
void bucle3(void) {
    unsigned long i;
    for (i=0;i<2000000;i++) {
        c=a*b*c; c=1/(a+b*c);
    }
}
```

# Salida del monitor `gprof`: *flat profile*

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
45.22	23.61	23.61	1	23.61	52.13	main
33.98	41.34	17.74	2	8.87	11.24	bucle1
16.32	49.86	8.52	18	0.47	0.47	bucle3
4.33	52.13	2.26	1	2.26	6.05	bucle2

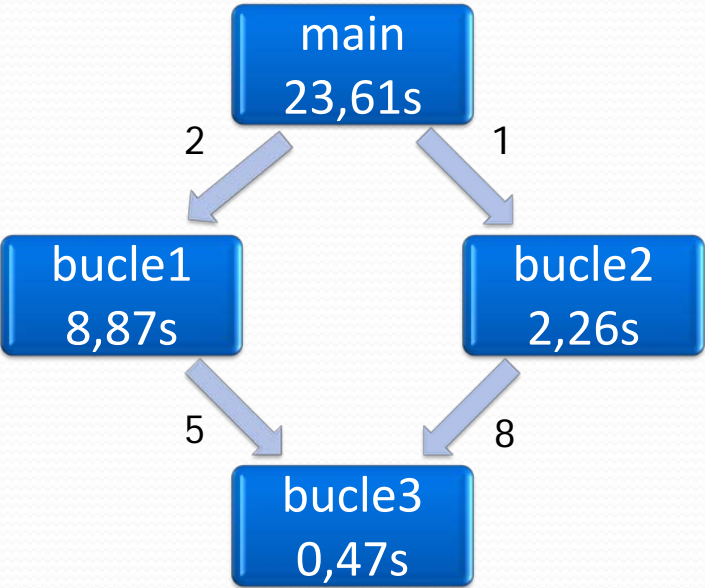


- **% time:** Tanto por ciento del tiempo total de CPU del programa que usa el código propio de la función (código propio es el que pertenece a la función y no a las funciones a las que llama).
- **Cumulative seconds:** La suma acumulada de los segundos consumidos (CPU) por el código propio de la función y por el de las funciones que aparecen encima de ella en la tabla.
- **Self seconds:** tiempo (CPU) total de ejecución del código propio de la función. Es el criterio por el que se ordena la tabla ( $Self\ seconds = calls \times self\ s/call$ ).
- **Self s/call:** tiempo (CPU) medio de ejecución del código **propio** por cada llamada a la función.
- **Total s/call:** tiempo (CPU) medio de ejecución de cada llamada a la función (contando tanto el tiempo del código propio como el de las funciones a las que llama).



# Salida del monitor gprof: call profile

call profile



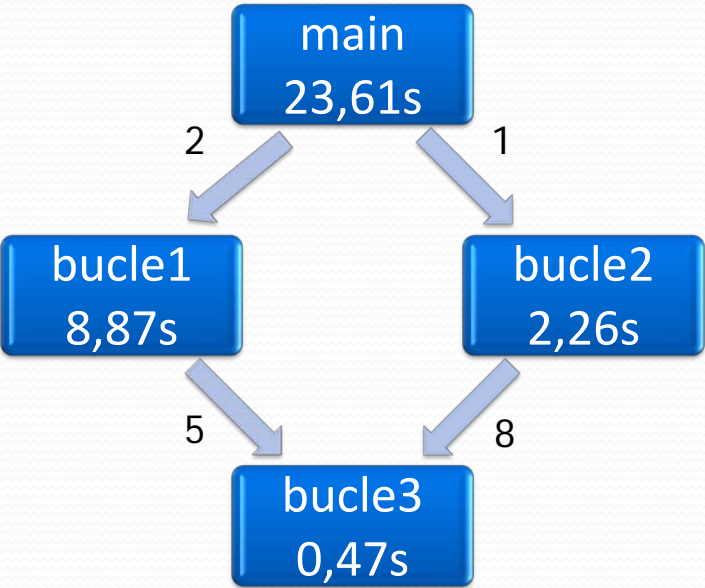
Grafo de llamadas  
(Call graph)

index	% time	self	children	called	name
[1]	100.0	23.61	28.52		main [1]
		17.74	4.73	2/2	bucle1 [2]
		2.26	3.79	1/1	bucle2 [4]
-----					
		17.74	4.73	2/2	main [1]
[2]	43.1	17.74	4.73	2	bucle1 [2]
		4.73	0.00	10/18	bucle3 [3]
-----					
		3.79	0.00	8/18	bucle2 [4]
		4.73	0.00	10/18	bucle1 [2]
[3]	16.3	8.52	0.00	18	bucle3 [3]
-----					
		2.26	3.79	1/1	main [1]
[4]	11.6	2.26	3.79	1	bucle2 [4]
		3.79	0.00	8/18	bucle3 [3]
-----					

bucle1 ha llamado a bucle3 10 de las 18 veces que es llamado bucle3 en total.

# Salida del monitor `gprof` call profile (cont.)

call profile



Grafo de llamadas  
(Call graph)

index	% time	self	children	called	name
[1]	100.0	23.61	28.52		main [1]
		17.74	4.73	2/2	bucle1 [2]
		2.26	3.79	1/1	bucle2 [4]
-----					
		17.74	4.73	2/2	main [1]
[2]	43.1	17.74	4.73	2	bucle1 [2]
		4.73	0.00	10/18	bucle3 [3]
-----					
		3.79	0.00	8/18	bucle2 [4]
		4.73	0.00	10/18	bucle1 [2]
[3]	16.3	8.52	0.00	18	bucle3 [3]
-----					
		2.26	3.79	1/1	main [1]
[4]	11.6	2.26	3.79	1	bucle2 [4]
		3.79	0.00	8/18	bucle3 [3]
-----					

De las 18 veces que *bucle3* es llamado, 8 proceden de *bucle2* y 10 de *bucle1*

# Otros profilers: Perf

- Perf es un conjunto de herramientas para el análisis de rendimiento en Linux basadas en eventos software y hardware (hacen uso de contadores hardware disponibles en los últimos microprocesadores de Intel y AMD). Permiten analizar el rendimiento de **a)** un hilo individual, **b)** un proceso + sus hijos, **c)** todos los procesos que se ejecutan en una CPU concreta, **d)** todos los procesos que se ejecutan en el sistema. Algunos de los *comandos* que proporciona:

```
usage: perf [--version] [--help] COMMAND [ARGS]
```

- **list**: Lista todos los eventos disponibles.
- **stat**: Cuenta el número de eventos.
- **record**: Recolecta muestras cada vez que se produce un determinado conjunto de eventos. Fichero de salida: perf.data.
- **report**: Analiza perf.data y muestra las estadísticas generales.
- **annotate**: Analiza perf.data y muestra los resultados a nivel de código ensamblador y código fuente (si está disponible).

# Perf: Algunos tipos de eventos

- **perf list**

• cpu-clock	[Sw]	• LLC-load-misses	[Hw]
• task-clock	[Sw]	• LLC-store-misses	[Hw]
• minor-faults	[Sw]	• dTLB-load-misses	[Hw]
• major-faults	[Sw]	• dTLB-store-misses	[Hw]
• context-switches OR cs	[Sw]	• dTLB-prefetch-misses	[Hw]
• cpu-cycles OR cycles	[Hw]	• iTLB-load-misses	[Hw]
• instructions	[Hw]	• branch-load-misses	[Hw]
• cache-misses	[Hw]	• sched:sched_stat_runtime	[Trace]
• branch-misses	[Hw]	• sched:sched_pi_setprio	[Trace]
• L1-dcache-load-misses	[Hw]	• syscalls:sys_exit_socket	[Trace]
• L1-dcache-store-misses	[Hw]	• [...]	

- La lista completa de eventos hardware depende del tipo concreto de microprocesador.

- Intel: <http://www.intel.com/Assets/PDF/manual/253669.pdf> (cap. 18,19).
- AMD: [http://support.amd.com/TechDocs/55072\\_AMD\\_Family\\_15h\\_Models\\_70h-7Fh\\_BKDG.pdf](http://support.amd.com/TechDocs/55072_AMD_Family_15h_Models_70h-7Fh_BKDG.pdf) (sección 3.24).

# Perf: Ejemplos de uso como profiler

- Ejemplo 1: Cuento el tiempo de CPU, número total de ciclos, instrucciones, cambios de contexto y fallos de página del proceso *matr\_multiplication*. Repito el experimento 3 veces:

```
> sudo perf stat -e task-clock,cycles,context-switches,page-faults,instructions,cache-misses \  
-r 3 ./matr_multiplication
```

```
Performance counter stats for './matr_multiplication' (3 runs):
```

10.536,91 msec	task-clock	#	0,998 CPUs utilized	( +- 0,34% )
24.550.524.279	cycles	#	2,330 GHz	( +- 0,34% )
923	context-switches	#	0,088 K/sec	( +- 0,51% )
2.974	page-faults	#	0,282 K/sec	( +- 0,01% )
25.209.901.316	instructions	#	1,03 insn per cycle	( +- 0,00% )
31.581.707	cache-misses			( +- 0,80% )

```
10,5534 +- 0,0364 seconds time elapsed ( +- 0,35% )
```

# Perf: Ejemplos de uso como profiler (II)

- Ejemplo 2: Recolecto muestras del programa `./bucles` cada cierto número de ocurrencias (por defecto 4000) del evento `task-clock`. Guardo la información en `perf.data`.

```
> sudo perf record -c 4000 -e task-clock ./bucles
```

```
[ perf record: Woken up 178 times to write data ]
```

```
[ perf record: Captured and wrote 44,305 MB perf.data (1410563 samples)]
```

Y ahora muestro las funciones de mi programa que más ciclos de reloj consumen:

```
> sudo perf report --stdio
```

```
# Samples: 1M of event 'task-clock'
```

```
# Event count (approx.): 5642252000
```

# Overhead	Command	Shared Object	Symbol
44.72%	bucles	bucles	[.] main
33.62%	bucles	bucles	[.] bucle1
17.34%	bucles	bucles	[.] bucle3
4.29%	bucles	bucles	[.] bucle2
0.00%	bucles	[kernel.kallsyms]	[k] __set_current_blocked

```
....
```



# Perf: Ejemplos de uso como profiler (II)

- Ejemplo 3: Muestro las funciones de mi programa que más ciclos de reloj consumen y las que más fallos de página provocan:

```
> sudo perf record -c 4000 -e task-clock,faults ./matr_multiplication
[ perf record: Woken up 70 times to write data ]
[ perf record: Captured and wrote 17,340 MB perf.data (442282 samples) ]
> sudo perf report --stdio
# Samples: 442K of event 'task-clock'
# Event count (approx.): 1769128000
# Overhead  Command          Shared Object      Symbol
 99.32%    matr_mult          matr_mult          [.] multiply_matrices
  0.34%    matr_mult          libc-2.13.so       [.] random
  0.15%    matr_mult          matr_mult          [.] initialize_matrices
...
# Samples: 63 of event 'page-faults'
# Event count (approx.): 807
# Overhead  Command          Shared Object      Symbol
 91.82%    matr_mult          matr_mult          [.] initialize_matrices
  6.44%    matr_mult          libc-2.13.so       [.] 0x000ca168
```

# Perf: Ejemplos de uso como profiler (III)

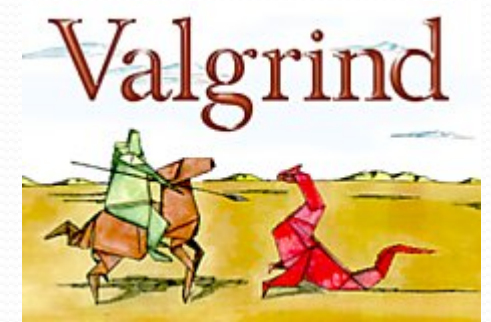
- Ejemplo 4: Muestro las líneas concretas de mi programa que más ciclos de reloj consumen:

➤ `sudo perf annotate -stdio`

Percent		Source code & Disassembly of matrix_multiplication
	:	<code>void multiply_matrices()</code>
	:	<code>...</code>
	:	<code>for (i = 0 ; i &lt; MSIZE ; i++) {</code>
	:	<code>for (j = 0 ; j &lt; MSIZE ; j++) {</code>
	:	<code>float sum = 0.0 ;</code>
	:	<code>for (k = 0 ; k &lt; MSIZE ; k++) {</code>
	:	<code>sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;</code>
	:	<code>...</code>
0.01	:	<code>12e8: movss (%rdx,%rax,1),%xmm0</code>
42.91	:	<code>12ed: mulss %xmm1,%xmm0</code>
10.69	:	<code>12f1: movss -0x4(%rbp),%xmm1</code>
2.16	:	<code>12f6: addss %xmm1,%xmm0</code>
9.22	:	<code>12fa: movss %xmm0,-0x4(%rbp)</code>

# Valgrind

- Valgrind es un conjunto de herramientas para el análisis y mejora del código de un programa (sólo disponible en Linux). Entre éstas, encontramos herramientas para la detección de errores de memoria (*memcheck*), detección de bloqueos o carreras en hebras (*hellgrind*), análisis del uso del heap (*massif*) y un profiler de memoria caché y de fallos de predicción de saltos (*cachegrind*).
- Valgrind analiza cualquier programa ya compilado (no necesita instrumentar el programa a partir de su código fuente).
- Valgrind actúa, esencialmente, como una máquina virtual que emula la ejecución de un programa ejecutable en un entorno aislado. Por tanto, la información suministrada es exacta y no estadística.
- Como desventaja, el sobrecoste computacional es muy alto. La emulación del programa ejecutable puede tardar decenas de veces más que la ejecución directa del programa de forma nativa.



# V-Tune (Intel) y CodeXL (AMD)

- Al igual que Perf, pueden hacer uso tanto de eventos software como hardware. Ambos programas funcionan tanto para Windows como para Linux, y permiten obtener información sobre los fallos de caché, fallos de TLB, bloqueos/rupturas del cauce, fallos en la predicción de saltos, cerrojos y esperas entre hebras, etc. asociados a cada línea del programa (tanto en código fuente como en código ensamblador).
- También se pueden usar como depuradores (debuggers), permiten la ejecución remota y son capaces de medir también el rendimiento de GPU, controlador de memoria, conexiones internas a la CPU, etc.

