Tema-3.pdf

Anónimo

Ingeniería de Servidores

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Tema 3

Monitorización de servicios y programas

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 /modificar/ sustituir/añadir (cuello de botella).
  - Qué parámetros del sistema hay que ajustar.
- Poder predecir cómo va a evolucionar la carga con el tiempo (capacity planning). Saber qué carga tengo en distintos momentos y así poder adelantarme en ciertos momentos donde pueda quedarse corto.
- Tarificar a los clientes (cloud computing). Medir el uso de los recursos.
- Obtener un modelo de un componente o de todo el sistema para poder deducir qué pasaría si...

Programador

- Conocer las partes críticas de una aplicación de cara a su optimización (hot spots). Identificar qué líneas de nuestro programa son responsables de que nuestro programa tarde más → para así cambiar esa parte del código.

Sistema Operativo

- Adaptarse dinámicamente a la carga. (modo turbo, prioridad de procesos…). Puntualmente puedo aumentar voltajes/frecuencias

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.) Lo que sufre el servidor.

Actividad de un servidor: conjunto de operaciones que se realizan en el servidor como consecuencia de la carga que soporta. Lo que yo puedo medir.

Algunas variables que reflejan la actividad de un servidor:

- ★ Procesadores: Utilización, temperatura, fCLK, 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

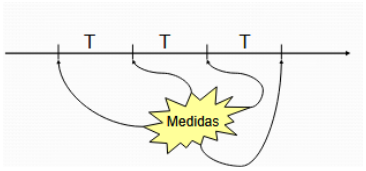
Herramienta diseñada para medir la actividad de un sistema informático y facilitar su análisis. Es una herramienta capaz de medir una de las variables anteriores.

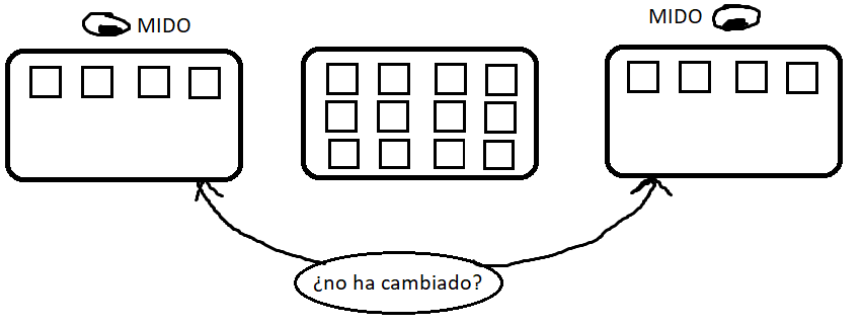
Acciones típicas de un monitor:

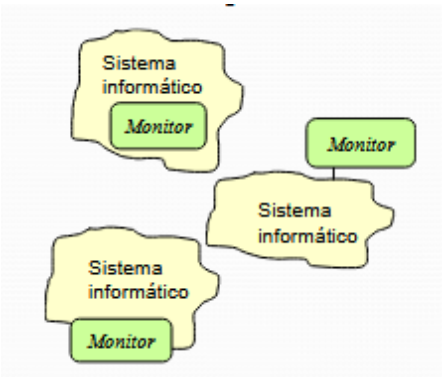
- Medir alguna/s variables que reflejen la actividad.
- Procesar y almacenar la información recopilada
- Mostrar los resultados

TIPOS DE MONITORES: ¿CÚANDO SE MIDE?

- Monitor por eventos (mide cada vez que ocurre un evento determinado)
  - Evento: cambio en el estado del sistema.
  - Mide el número de ocurrencias de uno o varios eventos
  - Información exacta
  - Limitados: solo miden ciertos eventos
  - Ejemplos de eventos:
    - Abrir/cerrar un fichero
    - Fallo en memoria cache
    - Interrupción de un dispositivo periférico
- Monitor por muestreo (cada cierto tiempo se despierta y mide):
  - Cada T segundos, siendo T el período de muestreo, el monitor realiza una medida.
  - La cantidad de información recogida depende de T
  - T puede ser, a su vez, variable
  - Información estadística. El período de muestreo me define cada cuanto hay que medir de forma que, al comparar el número de ficheros (por ejemplo), puede que no capte bien. No es exacto

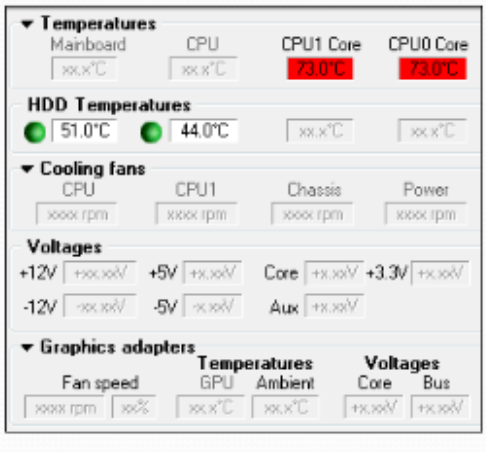






TIPOS DE MONITORES: ¿CÓMO SE MIDE?

- Software: Programas instalados en el sistema
- Hardware: Dispositivos físicos de medida (menor sobrecarga). No utilizan recursos de mi servidor. Por ejemplo.
  - *Sensor para medir la velocidad de rotación de los ventiladores*
  - Medidor de temperatura
- Híbridos: Utiliza los dos tipos anteriores. Mezcla entre Software y Hardware. Por ejemplo:
  - *Tengo que guardar lo medido en Hardware en algún sitio.*




```
$ 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
```

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)

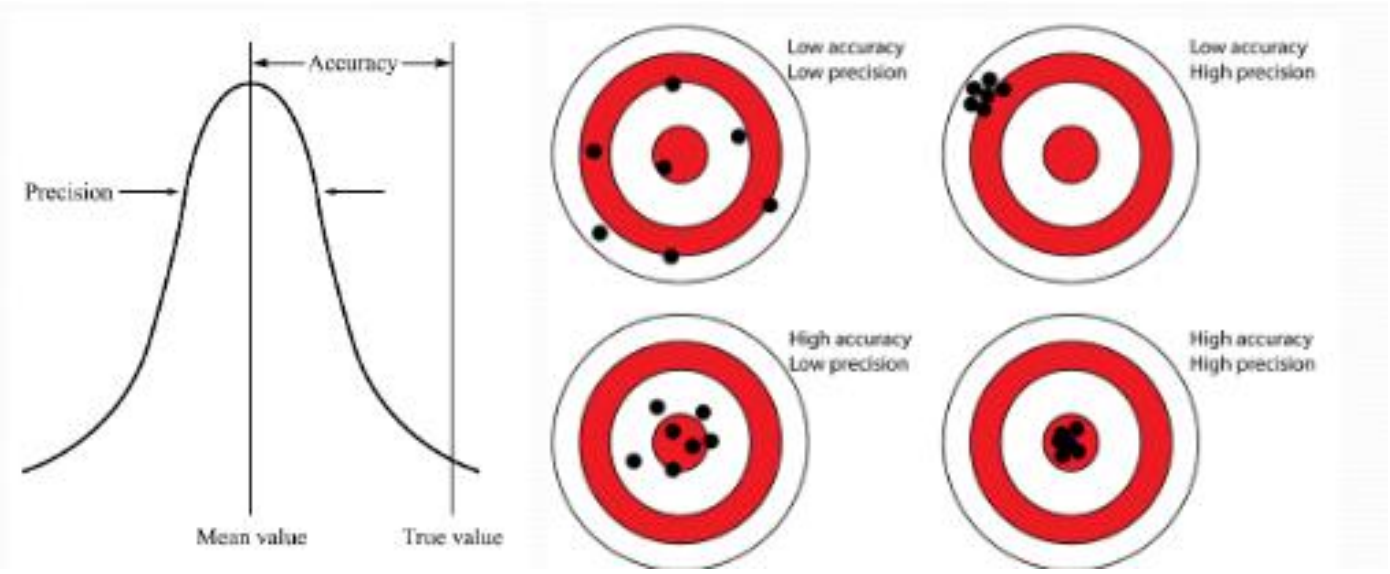
ATRIBUTOS QUE CARACTERIZAN A UN SENSOR/MONITOR

No es lo mismo exactitud que precisión.

- Exactitud de la medida (Accuracy, offset): ¿Cómo se aleja el valor medido del valor real que se quiere medir? Es la diferencia entre el valor real y el valor que mide → no sensor exacto
- Precisión (Precision): ¿Cuál es la dispersión de las medidas? Viendo la salida de mi sensor, comparo los valores que salen cuando tomo las mismas medidas. Cuanto más ancha la gráfica (la de abajo) peor
- Resolución del sensor: ¿Cuánto tiene que cambiar el valor a medir (lo que yo mido) para detectar un cambio?

Conceptos que se refieren al monitor:

- Tasa Máxima de Entrada (Max Input Rate): ¿Cuál es la frecuencia máxima de ocurrencia de los eventos que el monitor puede observar? (monitores por eventos). Puede ocurrir que los eventos lleguen tan rápidos que se les puede escapar alguno → Con qué frecuencia empieza a fallar.
- Anchura de Entrada (Input Width): ¿Cuánta información (p.ej. nº de bytes) se almacena por cada medida que toma el monitor? Cuando mide, cuántos bits cojo/almaceno por cada medida.



MAS ATRIBUTOS: SOBRECARGA (OVERHEAD)
Sobrecarga (overhead): ¿Qué recursos (%) le "roba" el monitor al sistema?
El instrumento de medida puede perturbar el funcionamiento del sistema.

Divido qué usa del recurso el monitor entre el recurso que tengo:

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

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

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

3.2. MONITORIZACIÓN A NIVEL DE SISTEMA

- A nivel general de los recursos del servidor.
- Recursos:
  - Núcleos
  - RAM
  - Almacenamiento (SSD, Discos duros…)
  - Conexiones de red
- Quiero ver cómo se usan esos recursos
- El SO es el que gestiona los recursos → se le preguntará a él

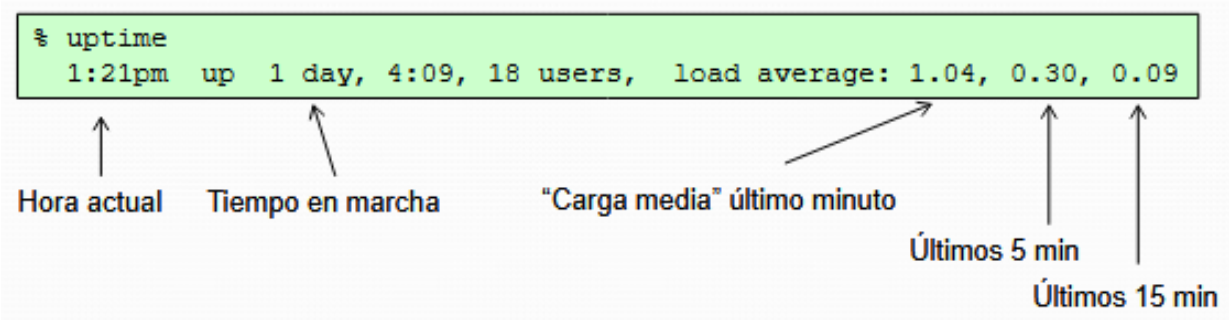
EL DIRECTORIO /PROC (LINUX)
Es una carpeta en DRAM utilizada por el núcleo de Linux para facilitar el acceso del usuario a la estructura de datos del SO. Es como si fuese una carpeta en la RAM, no en el almacenamiento. Sería absurdo porque cada vez que acceda tendría que entrar al disco duro → muy lento.

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

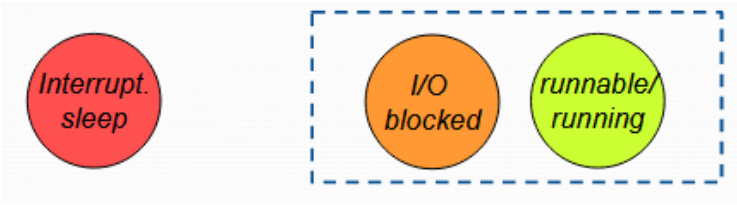
UPTIME
Tiempo que lleva el sistema en marcha y la "carga media" que soporta. Tiempo que la máquina lleva encendida independientemente de la carga → no me sirve mucho.
Me da información como el número de usuarios conectados y la carga media del sistema



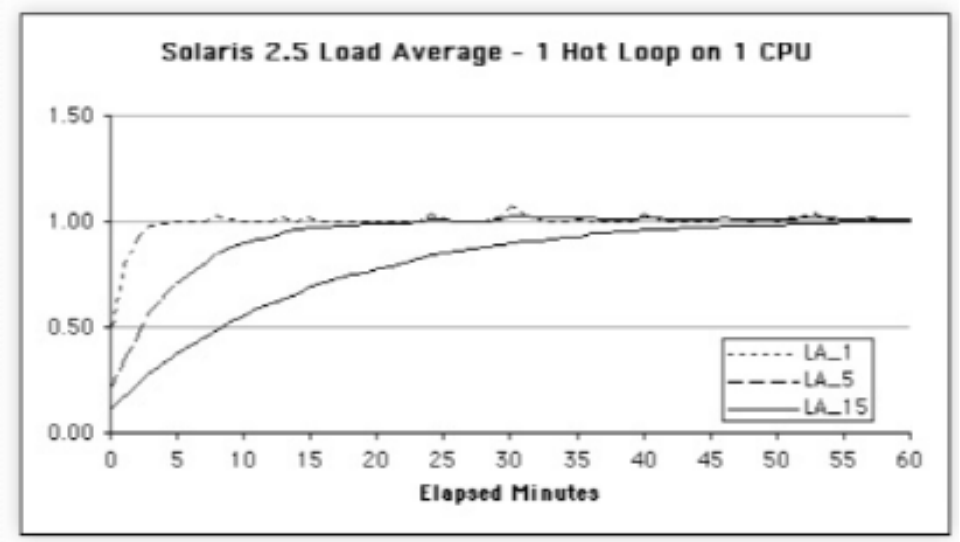
"CARGA" DEL SISTEMA SEGÚN LINUX
Estados básicos de un proceso:

- 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). Aunque puede consumir RAM/estar en memoria de intercambio, no se considera carga.

"Carga del sistema" (system load): número de procesos en modo running, runnable o I/O blocked
Carga = numero de procesos en un determinado estado (en ejecución/bloqueados)
UPTIME me dará una estimación de la carga.



¿CÓMO MIDE LA CARGA MEDIA EL SO?
Carga media: suma del numero de procesos durante un tiempo partido el número de muestras.
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· load(t) + (1-c)·LA(t-5) → **no es importante**

- 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 (c1>c5>c15). Si c = c1 calculamos LA_1(t), etc

PS (PROCESS STATUS)

Información sobre el estado actual de los procesos del sistema. Puedo ver qué procesos están ejecutándose (= Información más detallada de la carga) y en qué estado.

```
$ 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).

TOP

Muestra cada T segundos: carga media, procesos, consumo de memoria… Tiene predefinido el tiempo de muestreo. Se va actualizando cada poco tiempo.

Normalmente se ejecuta en modo interactivo (se puede cambiar T, las columnas seleccionadas, la forma de ordenar las filas, etc.)

La tercera línea nos dice el nivel de utilización de la CPU (wa: tiempo ocioso esperando E/S)

```
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 bdflush
   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)
Es un comando que se ejecuta de forma interactiva
Paging (paginación), swapping, interrupciones, cpu…
La primera línea no sirve para nada (info desde el inicio del sistema). De primeras me da un valor → a partir de ese ya podemos monitorizar.

```
% 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     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.

Ninguno de estos se usa → son interactivos. Prefiero que mida en 2º plano lo que yo quiero medir y que me lo guarde en el buffer de memoria. Hacen falta paquetes especiales:

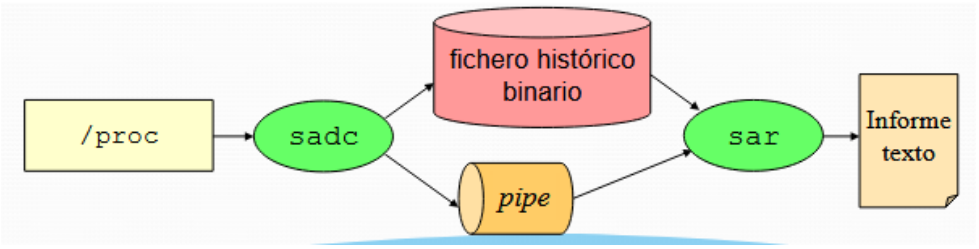
SYSSTAT

EL MONITOR SAR (SYSTEM ACTIVITY REPORTER)
Programa que monitoriza la actividad del servidor. Pertenece a un paquete especial diseñado para monitorizar la actividad del servidor. Lo hace preguntando lo que va a pasar. Existe modo interactivo, pero lo interesante es que guarda lo medido en un fichero.
Este monitor recopila información sobre la actividad del sistema:

- Actual: qué está pasando el día de hoy, o ahora mismo, en el sistema.
- Histórica: qué ha pasado en el sistema en otros días pasados.
    - Para ello utilizamos ficheros históricos en /var/log/sa/saDD donde los dígitos DD indican el día del mes (actualmente suele ponerse la fecha entera)

Esquema de funcionamiento:

- sadc (System-accounting data collector): Recoge los datos estadísticos (lectura de contadores) y construye un registro en formato binario (back-end). Hace la recolección de datos.
- sar: Lee los datos binarios que recoge sadc y las traduce a texto plano (front-end)



PARÁMETROS DE SAR

Hay una gran cantidad de parámetros. Puede funcionar en modo batch o en modo interactivo. Los parámetros importantes son los que están en negrita.

```
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 procesador (-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
...
```

-d → me desglosa la información por disco
-n → Información de la red
-f → le digo el día que quiero

Dos funcionalidades del SAR:

- Modo interactivo: cada cuánto tiempo quiero que me muestree durante cierto número de veces
- Modo no interactivo: Por defecto me va a dar la información de hoy (si pongo "sar" en terminal)

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 ...
```

Por defecto es "sar -u" cuando pongo "sar".

Utilización desglosada de cada CPU recopilada de forma interactiva cada 1s:

```
$ sar -P ALL 1
19:20:39 CPU  %usr  %nice %sys  %wa  %st   %idle
19:20:40 all  53.45  0.00 6.18  0.00 0.00  40.37
19:20:40   0  49.49  0.00 5.05  0.00 0.00  45.45
19:20:40   1  51.61  0.00 6.45  0.00 0.00  41.94
19:20:40   2  58.16  0.00 8.17  0.00 0.00  33.67
19:20:40   3  54.55  0.00 5.05  0.00 0.00  40.40
...
```

"-p ALL" es para información de cada CPU individualmente
"1" es para hacerlo en modo interactivo, ya que le estoy dando el tiempo de muestreo. Cada segundo me dará información

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/sa/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
```

-b → Información de los discos duros (cuántas lecturas por segundo hay, etc)
-d → Información de cada disco duro por separado

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  rxmcst/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
[…]
```

-n → Información de la red

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/sa
-rw-r--r--   1 root     root      3049952 Sep 30 23:55 sa30
-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

Anchura de entrada: de media cuántos Bytes de información escribe en el fichero cada vez que se despierta.

Datos de partida:
Extracto de `ls /var/log/sa`:
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.

```
-rw-r--r--   1 root root 3049952  Oct 2  23:55  sa02
```

Fichero sa02 (día 2 de octubre)

Vemos que el fichero se ha escrito por completo (23:55). Al final del dóa se han escrito 3.049.952 bytes.

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.

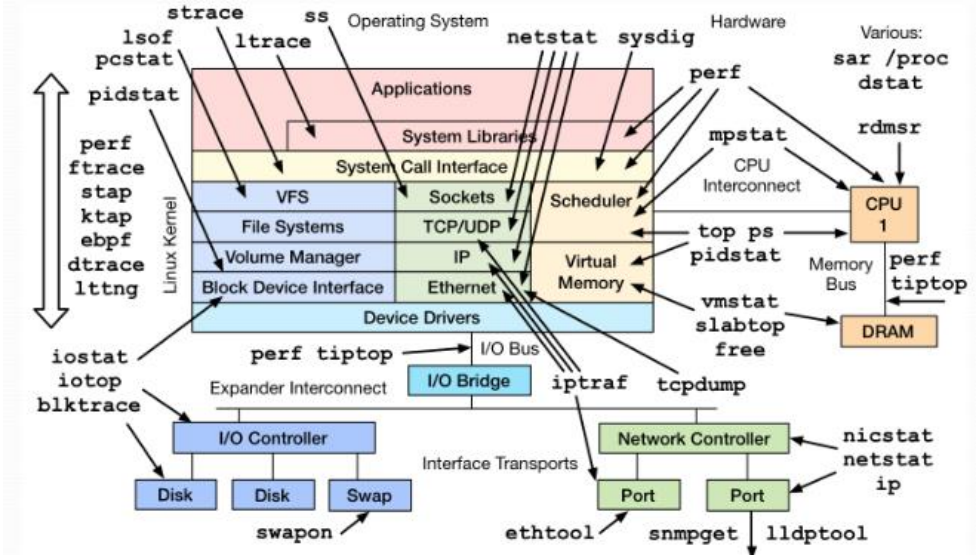
- Cada hora se recogen 12 muestras.
- Al día se recogen 24 x 12 = 288 muestras.

Anchura de entrada del monitor:

- Cada registro ocupa, de media, 10590,1 bytes (aproximadamente 10,59 KB o 10,34KiB)

OTRAS HERRAMIENTAS PARA MONITORIZACIÓN

Todas estas al final preguntan al SO. Dependiendo de lo que necesitemos usaremos unas u otras.



Una herramienta más grande se instala. Las demás lo hacen como plugins. Esto permite controlar más de un servidor a la vez.

CollectL: http://collectl.sourceforge.net/. Parecido a sar. Es capaz de ejecutarse de forma interactiva o como un servicio/demonio para recopilar datos históricos.
Nagios: https://www.nagios.org/. Permite la monitorización y la generación de alarmas de equipos distribuidos en red (tanto recursos HW como servicios de red). Se puede personalizar mediante la programación de plugins propios.
Otras herramientas que permiten la monitorización de equipos distribuidos en red: Ganglia, Munin, Zabbix, Pandora FMS...

PROCEDIMIENTO SISTEMÁTICO DE MONITORIZACIÓN: MÉTODO USE (UTILIZACIÓN, SATURACIÓN, ERRORS)

Con tantas herramientas y funcionalidades puedo volverme loco. ¿Qué hago para solucionar X problema? ¿Cuál uso?

Existe el método USE. No me vuelvo loco con 20.000 herramientas. Me pregunto: ¿qué tengo en mi servidor? RAM, Discos duros, CPU, Red.
Para cada uno de estos recursos comprobamos/medimos:

- Utilización: (en sar: -u) % de utilización del recurso (tiempo de CPU, tamaño de memoria, ancho de banda de E/S de cada disco o de cada tarjeta de red, etc)

- Saturación: Ocupación de las colas de aquellas tareas que quieren hacer uso de ese recurso (en el caso de la memoria, cantidad de swapping a disco). ¿Hay colas?
  - Importancia del valor medio. *En el comedor de la facultad puede que durante la mayor parte del tiempo no haya cola pero en una hora determinada venga mucha gente y se forme una gran cola.*

- Errores: Mensajes de error del kernel sobre el uso de dichos recursos. *Por ejemplo, fallos de página.*

3.3. MONITORIZACIÓN A NIVEL DE APLICACIÓN (PROFILERS)

Objetivo de un profiler: Monitorizar la actividad generada por una aplicación concreta con el fin de obtener información para poder optimizar su código.

Intenta buscar culpables → busca culpable de una actividad con un proceso concreto con el fi de optimizar el código o funcionamiento de mi programa. ¿QUIÉN HA GENERADO EL PROBLEMA?

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)?
- ¿Cuántas veces se ejecuta cada línea de programa?
- ¿Cuántas veces se llama a un procedimiento y desde dónde?
- ¿Cuánto tiempo tarda en ejecutarse (el código propio de) un procedimiento?
- ¿Cuántos fallos de caché/página genera cada línea del programa?
- ¿Qué cantidad de memoria está ocupada en cada momento del programa?
- Etc

- Ejemplo: CPU
  - Utilización: vmstat 1, "us" + "sy" + "st"; sar -u, sumando todos los campos excepto "%idle" e "%iowait"; ...
  - Saturación: vmstat 1, "r" > nº de CPU; sar -q, "runq-sz" > nº de CPU;...
  - Errores: usando *perf* si la opción de "eventos de errores específicos de CPU" está habilitada; ...

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. Me da información muy básica sobre qué esta pasando con el programa.

No confundir con la orden TIME. En esta ocasión hay que poner el path completo: "/usr/bin/time".

Información que da:

- User time: tiempo de CPU ejecutando en modo usuario.
- System time: tiempo de CPU ejecutando código del núcleo.
- Elapsed (wall clock) time: tiempo que tarda el programa en ejecutarse. Tiempo que ha pasado entre el inicio y el final(entre medias puede haber ejecutado otros procesos)
- Major page faults: fallos de página que requieren acceder al almacenamiento permanente para obtener la información de estos.
- Minor page faults: fallos de página que no requieren acceder al almacenamiento permanente porque ya está en la RAM.
- 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".

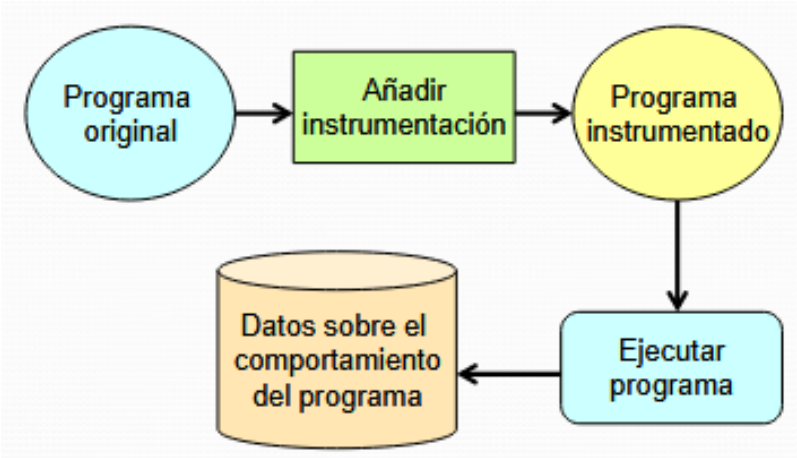
```
% /usr/bin/time -v ./matr_mult2
User time (seconds): 4.86
System time (seconds): 0.01
Percent of CPU this job got: 99%
Elapsed (wall clock) time 0:04.90
Maximum RSS (kbytes): 48784
Major page faults: 0
Minor page faults: 3076
Voluntary context switches: 1
Involuntary context switches: 195
...

No confundir con:
% time  ./matr_mult2
real    0m4.862s
user    0m4.841s
sys     0m0.010s
```

PROFILER GPROF → Dentro del compilador

Estima el tiempo de CPU que consume cada función de un proceso/hilo. Es decir, su objetivo es dar información sobre el tiempo de CPU que consume el programa. Se centra en eso, nada más).. También calcula el número de veces que se ejecuta cada función y cuántas veces una función llama a otra. Es el primer programa al que podemos llamar profiler.

Utilización de gprof para programas escritos en C, C++:

- Instrumentación en la compilación:
  - Gcc prog.c -o prog -pg -g
    - -pg cambia cosas del programa dándome un ficheron para ver la información.
- Ejecución del programa y recogida de información:
  - ./prog
  - La información recogida se deja en eñ 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 incluye la de las funciones de las bibliotecas con las que se enlaza el 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 la función (cada vez que se produce el evento de inicio de la función) y c2 para estimar el tiempo de CPU/ejecución de la función. Mete código para ello.
- El S.O. programa un temporizador (por defecto se inicializa a 0,01s) que se decrementará cada vez que se ejecute código del programa (cada vez que el SO pone el programa en la CPU). Cuando pasen 0.01 segundos mido: ¿por dónde está? Así mido el tiempo de ejecución. Cada 0.01s de tiempo de ejecución del programa se incrementa el contador.

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.

Solamente se interrumpe el programa cada 0.01s de tiempo de ejecución del programa.
Si mi programa tarda 40 segundos de ejecución, puede tardar realmente más porque puede acceder a ficheros. Por tanto solo mide el tiempo de CPU, no el tiempo real aunque sea el único proceso del sistema.

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

Suponemos que tenemos un programa muy sencillo: bucles.c.
Instrumentación (-pg) en la compilación: gcc bucles.c -pg –g -o bucles
Ejecución del programa monitorizado: ./bucles
Obtención de la información recopilada: gprof bucles

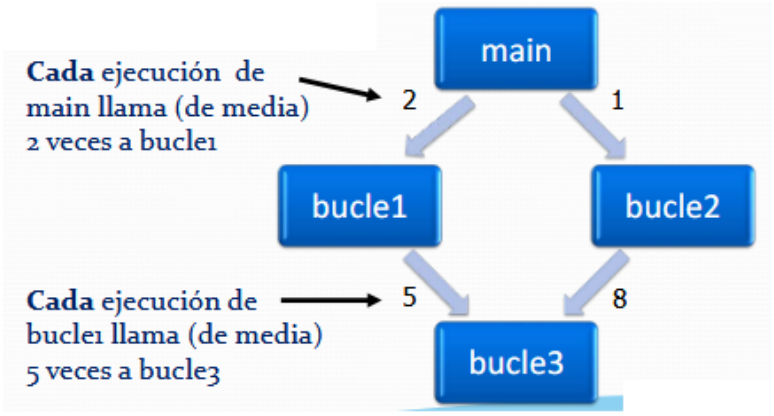
Tiene 4 funciones:

- main llama dos veces a bucle1 y una vez a bucle 2
- bucle1 tiene un bucle bien gordo y llama 5 veces al bucle 3
- bucle 2 tiene un bucle bien gordo y llama 8 veces al bucle 3
- bucle3 tiene un bucle bien gordo.

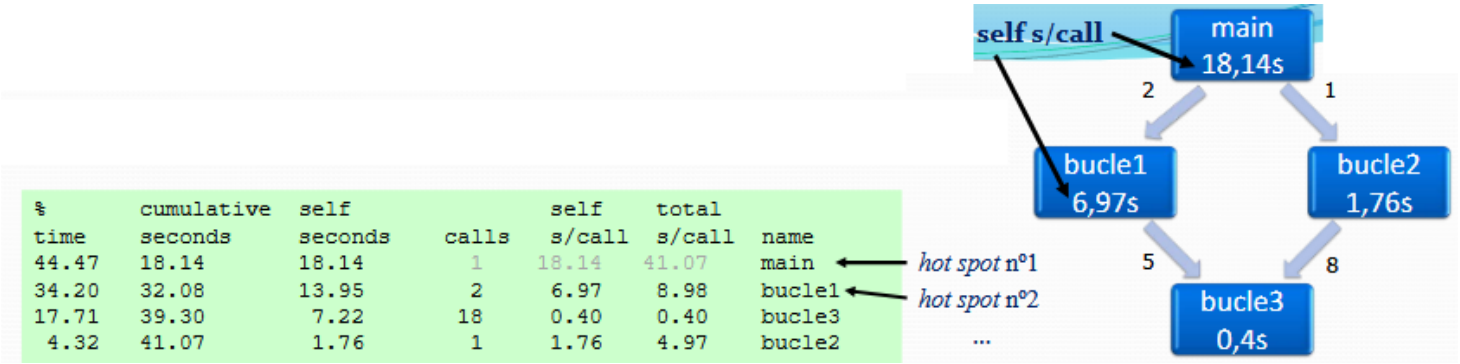
bucles.c

```
float a=0.3; float b=0.8; float c=0.1;
void main(void) {
  unsigned long i;
  for (i=0;i<80000000;i++) a=a*b/(1+c);
  bucle1(); bucle1(); bucle2();
}
void bucle1(void) {
  unsigned long i;
  for (i=0;i<20000000;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<5000000;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<1000000;i++) {
    c=a*b*c; c=1/(a+b*c);
  }
}
```

¿Cuál es la función que más tiempo de CPU consume?



SALIDA DEL MONITOR GPROF: FLAT PROFILE



% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name	
44.47	18.14	18.14	1	18.14	41.07	main	← *hot spot nº1*
34.20	32.08	13.95	2	6.97	8.98	bucle1	← *hot spot nº2*
17.71	39.30	7.22	18	0.40	0.40	bucle3	
4.32	41.07	1.76	1	1.76	4.97	bucle2	
						...	

Muestra una tabla con todas las funciones que tengo en el programa ordenadas según el tiempo del código propio de cada función.

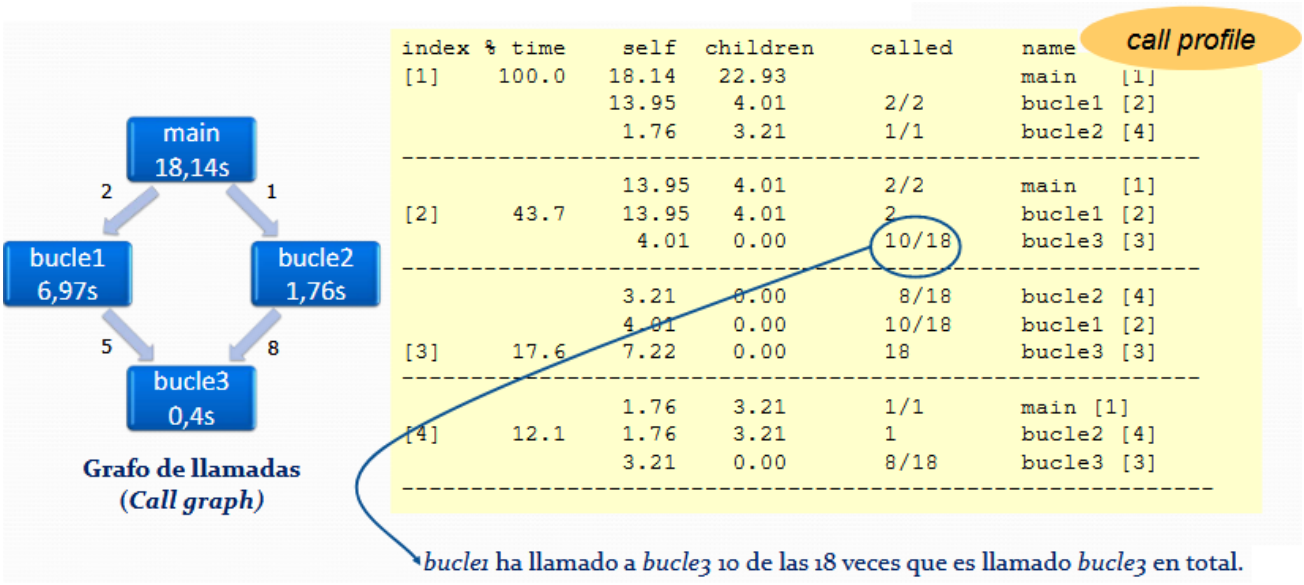
Tiempo del bucle 2: código propio del búcle 2 + 8 veces código propio del bucle 3.

Si tuviese E/S GProf no se entera → Solo mide el tiempo de CPU que consume mi programa.
A la primera función de la tabla se le llama hot spot nº1 → Es la que habría que optimizar.

- % time: Tanto por ciento del tiempo total de CPU del programa que usa el código propio de la subrutina (código propio es el que pertenece a la subrutina y no a las subrutinas a las que llama).
- Cumulative seconds: La suma acumulada de los segundos consumidos (CPU) por la subrutina y por las subrutinas que aparecen encima de ella en la tabla (código propio). *La información en gris en la tabla no aparece (p.e en la fila del main).* La única forma de ver el TCPU del programa es ver el último valor de esta columna.
- Self seconds: tiempo (CPU) total de ejecución del código propio de la subrutina. Es el criterio por el que se ordena la tabla (Self seconds = calls × self s/call). Cuánto tiempo de CPU consume el código propio de cada función → no por cada llamada sino en total. Es la columna más importante
- Self s/call: tiempo (CPU) medio de ejecución del código propio por cada llamada a la subrutina. (segundos por llamada). Cuánto tiempo tarda el código propio. No es la definitiva porque tengo que tener en cuenta cuántas veces se llama a cada función.
  - Bucle 3 → se le llama 18 veces (2*5+8)
- Total s/call: tiempo (CPU) medio total de ejecución por cada llamada a la subrutina, es decir, contando las subrutinas a las que ésta llama. Es decir, cuanto tiempo pasa desde que llamo a la función hasta que sale/retorna.

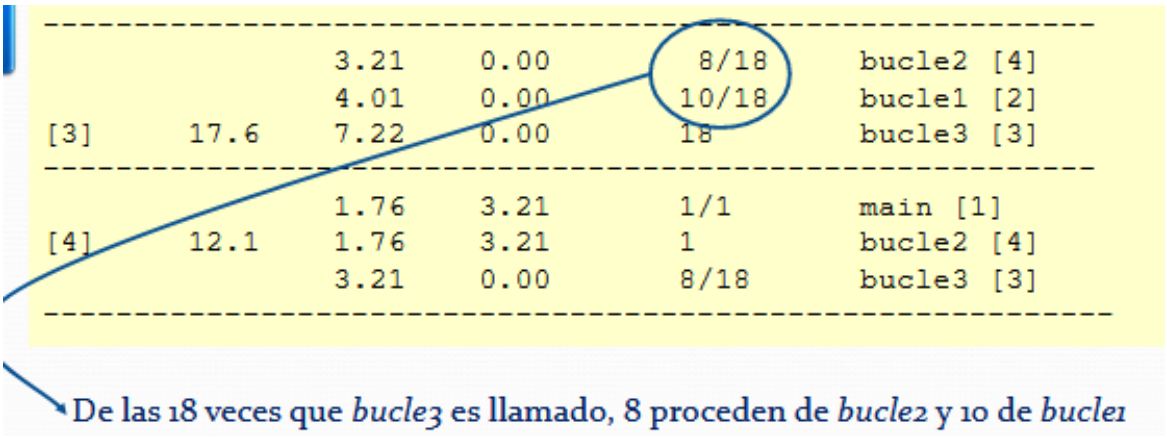
Cualquier programa que se llame profiler, me tiene que dar al menos el flat profile.
Con este programa me da 4107 muestras. De ellas, el 44,47% han caído en el main.
Como es un muestreo, si ejecuto gprofile dos veces, no me dará los mismos resultados. Es una estimación, los tiempos son estimados. Sin embargo, el número de llamadas no va a cambiar.
El contador c2 es el que se incrementa si cuando paro cada 0,01s estoy en esa funcion → Así es como estimo.

SALIDA DEL MONITOR GPROF: CALL PROFILE



LO IMPORTANTE: LAS DOS ULTIMAS COLUMNAS.
El procedimiento:
Miro la columna index. Trazo línea horizontal y veo a qué funcion se refiere.
Por ejemplo, la fila [2] hace referencia a bucle 1 → Lo que están por debajo son las funciones a las que llama bucle 1 (bucle 3) y lo de arriba quién llama a bucle 1 (el main)

Los números de la columan "called" significan:
El de la línea horizontal al índice es el número de veces que la función es llamada (*por ejemplo el bucle 1 es llamado 2 veces)*
Los de arriba son el número de veces de esas que es responsable cada función de llamar a la función a la que hace referencia la fila (*por ejemplo, el main es el responsable de las dos únicas llamadas al bucle 1).*
Los de abajo son el número de ves que llama la funci´´on a la que hace referencia la fila a cada función del total. (*por ejemplo, bucle 1 llama 10 veces al bucle 3, de las 18 veces que bucle 3 es llamada)*



MONITOR GCOV

Aporta información sobre el número de veces que se ejecuta cada línea de código del programa.
Utilización de gcov

- Instrumentación en la compilación
  - gcc prog.c –o prog –fprofile-arcs –test-coverage
- Ejecución del programa y recogida de información
  - ./prog
  - La información recogida se deja en varios ficheros
- Visualización de la información referida a la ejecución del programa
  - gcov prog.c (genera prog.c.gcov)

prog.c

```
            void main() {
         1    producto();
         1    producto();
         1    producto();
         1    division();
         1    division();
         1  }

           producto() {
 150000003   for (i=0;i<50000000;i++)
 150000000    c=a*b;
         3  }

           division() {
  60000002   for (i=0;i<30000000;i++)
  60000000    c=a/b;
         2  }
```

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.

- ★ No necesita el código fuente (como ventaja).
- ★ Puede monitorizar un programa arrancado.
- ★ No hace falta que sea un hilo, puede monitorizar un proceso con todos sus hijos
- ★ No hace falta que esté en c++
- ★ No se puede utilizar en Máquina Virtual

GProf utilizaba un temporizador de 0.01s y tomaba una muestra cada 0.01s (se decrementaba el temporizador en 0.01s y, cuando el tiempo llegaba a 0, se tomaba una muestra). Perf hace esto y muchos otros eventos más.
Algunos de los comandos que proporciona:

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

- ➢ list: Lista todos los eventos disponibles.
- ➢ stat: Cuenta el número de eventos. Puedo ver cuántos eventos han saltado (parecido a usr/bin/time)
- ➢ 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 list (lista de eventos):
Task-clock es como gprof.

```
• 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]  • [...]
```

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 `noploop`. Repito el experimento 10 veces

> perf stat -e task-clock,cycles,context-switches,page-faults,instructions,cache-misses -r 10 ./noploop

```
Performance counter stats for './noploop' (10 runs):

      1.024,54  msec task-clock                #    0,999 CPUs utilized            ( +- 0,03% )
  2.386.656.279  cycles                      #    2,329 GHz                      ( +- 0,01% )
             93  context-switches            #    0,090 K/sec                    ( +- 2,77% )
             44  page-faults                 #    0,043 K/sec                    ( +- 0,84% )
  1.022.192.244  instructions                #    0,43  insn per cycle           ( +- 0,00% )
          9.105  cache-misses                                                    ( +- 10,90% )

    1,026042 +- 0,000444 seconds time elapsed  ( +-  0,04% )
```

Con perf también puede recolectar muestras de un programa → perf record:
Ejemplo 2: Recolecto muestras del programa ./naive cada cierto número de ocurrencias (por defecto 4000) de los eventos: cpu-clock y faults. Guardo la información en perf.data.

> perf record -c 4000 -e cpu-clock,faults ./naive

```
[ perf record: Woken up 14 times to write data ]
[ perf record: Captured and wrote 3.44 MB perf.data (~150220 samples) ]
```

Con el comando le estoy pidiendo 4000 muestras de eventos de tipo cpu-clock y faults.

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

> perf report --dsos=naive,libc-2.13.so

```
# cmdline : /usr/bin/perf_3.6 record -e cpu-clock,faults ./naive
# Samples: 74K of event 'cpu-clock'
# Event count (approx.): 74813
# Overhead  Command      Shared Object      Symbol
  99.32%     naive        naive              [.] multiply_matrices
   0.34%     naive        libc-2.13.so       [.] random
   0.15%     naive        naive              [.] initialize_matrices
...
# Samples: 63  of event 'page-faults'
# Event count (approx.): 807
# Overhead  Command      Shared Object      Symbol
  91.82%     naive        naive              [.] initialize_matrices
   6.44%     naive        libc-2.13.so       [.] 0x000ca168
```

GPROF me da un caso muy particular de PERF. PERF tiene una última opción donde se puede descender al código ensamblador → perf annotate → me dice qué líneas han causado el mayor evento (Tcpu, fallos de página…) de mi programa

Ejemplo 4: Muestro las líneas concretas de mi programa que más ciclos de reloj consumen:
> perf annotate --stdio --dsos=naive --symbol=multiply_matrices

```
Percent |      Source code & Disassembly of naive
        :      void multiply_matrices()
        :      ...
        :        for (i = 0 ; i < MSIZE ; i++) {
        :          for (j = 0 ; j < MSIZE ; j++) {
        :            float sum = 0.0 ;
        :            for (k = 0 ; k < MSIZE ; k++) {
   1.67 :      8c18:       add     r3, r3, #1
  54.13 :      8c1c:       cmp     r3, #500        ; 0x1f4
        :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
   2.39 :      8c20:       mov     r9, r1
  36.60 :      8c24:       vmla.f32        s15, s13, s14
        :      ...
```

Hasta ahora hemos visto una forma de funcionar: cada vez que ocurra x cosa mido.

VALGRIND

Valgrind es un conjunto de herramientas para el análisis y mejora del código.
Entre éstas, encontramos:

- Callgrind, una versión más refinada de gprof.
- Cachegrind, un profiler de la memoria caché.
- Memcheck, un detector de errores de memoria.

Valgrind puede analizar cualquier binario 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. Valgrind coge el ejecutable y lo traduce a su código propio (con su lenguaje), lo instrumentaliza (lo modifica) según lo que quiero hacer, lo vuelve a compilar y lo ejecuta → A costa de que el programa dure mucho más. No es recomendable para programas de tiempo real. Por eso no me va a dar el Tcpu, pero para otras cosas es muy interesante.
Por tanto, 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)

Suelen utilizarse como back-end (Para el front-end se utilizan los IDE) → Tengo mi entorno de desarrollo, lo compilo, lo depuro y además puedo ver información de mi programa y hacerles profiling.
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

Skiding (?): en mi procesador se ejecutan muchos programas en paralelo. ¿Cuál ha causado el error? PERF no puede detectarlo. Estos (V-Tune y CodeXL) sí.