Tema-3-Monitorizacion-de-servici...

fer__luque

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

1. Concepto de monitor de actividad

¿Para qué monitorizar un servidor?

El administrador/ingeniero debe conocer cómo se usan los recursos para:

- Saber qué hardware hay que reconfigurar/sustituir/añadir y qué parámetros del sistema hay que ajustar
- Predecir cómo va a evolucionar la **carga** con el tiempo (*capacity planning*)
- Tarificar a los clientes (*cloud computing*)
- Obtener un modelo de un componente o de todo el sistema para poder deducir qué pasaría si...

El programador debe conocer también las partes críticas (*hot spots*) de una aplicación de cara a su optimización.

El sistema operativo debe ser capaz de adaptarse dinámicamente a la carga.

La carga y la actividad de un servidor

- Por **carga** (*workload*) se entiende el conjunto de tareas que ha de realizar el servidor, es decir, todo aquello que demande sus recursos
- Por **actividad** de un servidor se entiende el conjunto de operaciones que se realizan en el servidor como consecuencia de la carga que soporta

Entre las variables que reflejan la actividad de un servidor encontramos:

- Procesador: utilización, temperatura, frecuencia, procesos, interrupciones...
- Memoria: accesos, memoria usada, fallos de caché, fallos de página...
- Discos: lecturas/escrituras por tiempo, longitud de las colas...
- Red: paquetes recibidos/enviados, colisiones por segundo...
- Sistema global: nº usuarios, nº peticiones...

Definición de monitor de actividad

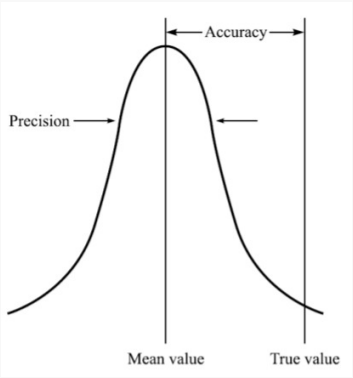
Por **monitor de actividad** entendemos la herramienta diseñada para medir la actividad de un sistema informático y facilitar su análisis. Se encargan de medir algunas variables, procesar y almacenar información y mostrar resultados

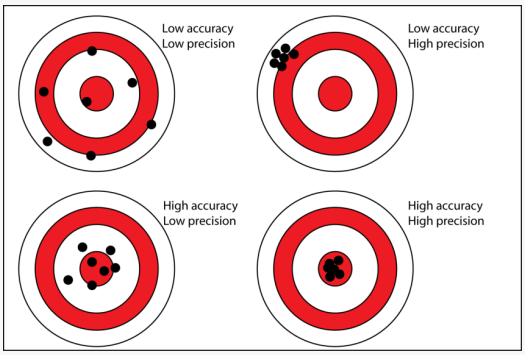
Tipos de monitores:

- **¿Cuándo se mide?**
  - **Monitor por eventos**: cada vez que ocurre un evento (cambio en el sistema)
    - Aporta información exacta
  - **Monitor por muestreo**: cada cierto tiempo T (período de muestreo)
    - Aporta información estadística
- **¿Cómo se mide?**
  - **Software**: programas instalados en el sistema
  - **Hardware**: dispositivos físicos de medida (menor sobrecarga)
  - **Híbridos**: mezclan ambos
- **¿Existe interacción con el analista/administrador?**

Atributos que caracterizan a un sensor/monitor

- **Exactitud** de la medida (*Accuracy*, *offset*): se refiere a cómo se aleja el valor medido del valor real que se quiere medir
- **Precisión** (*Precision*): cuál es la dispersión de las medidas
- **Resolución** del sensor: cuánto tiene que cambiar el valor a medir para detectar un cambio





- **Tasa máxima de entrada** (*Max Input Rate*): cuál es la frecuencia máxima de los eventos que el monitor puede observar
- **Anchura de Entrada** (*Input Width*): cuánta información se almacena por cada medida que toma el monitor
- **Sobrecarga** (*Overhead*): qué recursos le "roba" el monitor al sistema:
  $Sobrecarga_{Recurso}(\%) = \frac{Uso\ del\ recurso\ por\ el\ monitor}{Capacidad\ total\ del\ recurso} \times 100$

2. Monitorización a nivel de sistema

El directorio /proc (Linux)

Es una carpeta en DRAM utilizada por el kernel para facilitar el acceso del usuario a las estructuras de datos del SO

A traves de /proc podemos:

- Acceder a información global sobre el SO
- Acceder a la información de cada uno de los procesos del sistema (*/proc/[pid]*)
- Acceder y, a veces, modificar algunos parámetros del kernel del SO (*/proc/sys*)

En Linux, la mayoría de los monitores de actividad a nivel de sistema usan como fuente de información este directorio

uptime

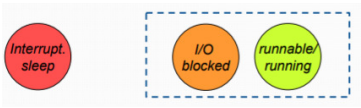
Nos dice el tiempo que lleva el sistema en marcha y la "carga media" que soporta.

Carga del sistema según Linux

Estados básicos de un proceso:

- En ejecución (*running*) o esperando que haya un núcleo (*core*) libre para ser ejecutado (*runnable*). La cola de procesos está formada por aquellos que se están ejecutando y los que pueden ejecutarse
- Bloqueado esperando a que se complete una operación de E/S para continuar
- Durmiendo esperando a un evento del usuario o similar

Por "carga del sistema", Linux entiende el número de procesos en modo *running*, *runnable* o *I/O blocked*



¿Cómo mide la carga media el SO?

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

Es decir, el valor se actualiza cada 5 segundos, siendo c la influencia en la carga media de la carga actual

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

ps (*process status*)

Nos da información sobre el estado actual de los procesos del sistema

top

Muestra cada T segundos la carga media, los procesos, el consumo de memoria.

Normalmente se ejecuta en modo interactivo

vmstat (*virtual memory statistics*)

Nos da información sobre el *paging*, *swapping*, inerrupciones, cpu...

Con otros argumentos, puede dar información sobre acceso a discos (partición de swap) y otras estadísticas de memoria.

Sysstat

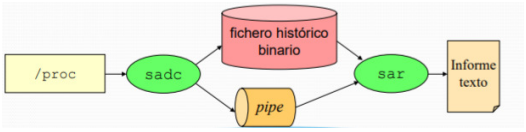
El monitor *sar* (*system activity reporter*)

Recopila información sobre la actividad del sistema:

- **Actual**: qué está pasando ahora mismo o en el día de hoy en el sistema
- **Histórica**: qué ha pasado en días pasados (ficheros en */var/log/sa/saDD*)

Esquema de funcionamiento:

- *sdac* (*system-accounting data collector*): es el que recoge los datos estadísticos (lectura de contenedores) y construye un registro en formato binario
- *sar*: lee los datos que recoge *sadc* y las traduce a texto plano (*front-end*)



Parámetros de *sar*

Muchos parámetros (que pueden funcionar en modo batch o en modo interactivo)

Conocer: -f, -s, -e, -u, -P, -b, -d, -n

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.

Cada ejecución de *sadc* añade un registro binario con los datos recogidos al fichero histórico del día

Cálculo de la anchura de entrada del monitor

Datos de partida

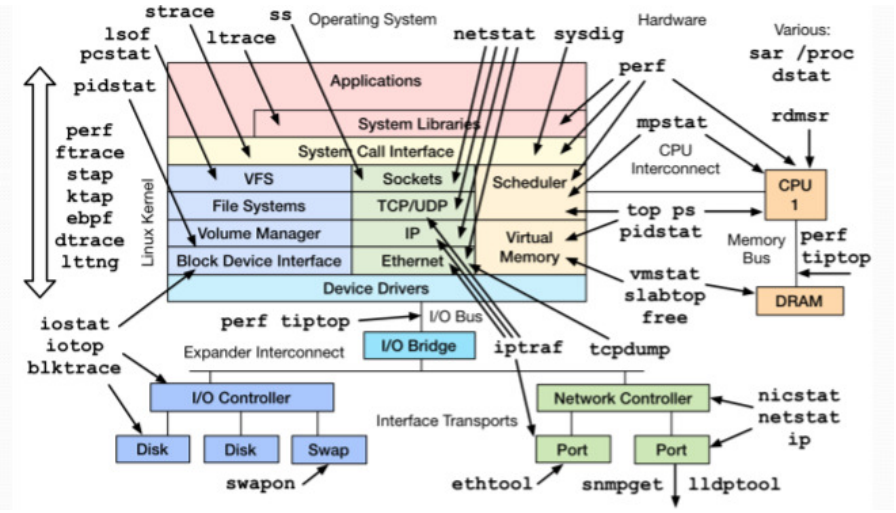
Vemos que cada fichero histórico de un día ocupa 3049952 bytes, y suponemos que la primera muestra se toma a las $00:00$ de cada día y que *sadc* se ejecuta con un tiempo de muestreo constante:

Solución

- El fichero ocupa 3049952 bytes
- *sadc* se ejecuta cada 5 minutos
  - Cada hora se recogen 12 muestras
  - Al día se recogen $24 \times 12 = 288$ muestras
- Por lo que:

  Anchura de entrada $= \frac{3049952\ bytes}{288\ muestras} = 10590,1\ bytes$, aproximadamente $10,59KB\ (K = 10^2)$ o $10,34KiB\ (Ki = 2^{20})$

Otras herramientas para monitorización



- **CollectL**: Parecido a sar. Interactivo o servicio/demonio para recopilar datos históricos
- **Nagios**: Monitorización y generación de alarmas de equipos distribuidos en red. Permite programar plugins para personalización. Más herramientas para equipos en red: Ganglia, Munin, Zabbix, Pandora FMS

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

Para cada recurso (CPU, memoria, E/S, red, ...) comprobamos:

- **Utilización**: Tanto por ciento de utilización del recurso
- **Saturación**: Ocupación de las colas de aquellas tareas que quieren hacer uso de ese recurso
- **Errores**: Mensajes de error del kernel sobre el uso de dichos recursos (fallo de caché, falta de página, ...)

Nota

Un porcentaje de utilización bajo no significa que no haya colas, ya que el porcentaje de utilización es un valor medio, ha estado mucho tiempo "parado", y después se le ha solicitado la ejecución de muchos procesos de forma simultánea.

3. Monitorización a nivel de aplicación (*profilers*)

Con el fin de que los monitores sean también herramientas útiles para los programadores, se desarrollan los ***profilers***, que monitorizarán 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*:

- Tiempo de ejecución de un programa y divisiones (CPU, E/S, ...)
- En qué parte pasa más tiempo (**hot spots**)
- Cuántas veces se ejecuta cada línea
- Cuantas veces se llama a un procedimiento y desde donde
- Fallos de caché/página de cada línea
- ...

Una primera aproximación: `/usr/bin/time`

No confundir con la orden `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
- `Elapsed (wall clock) time`: tismpo que tarda el programa en ejecutarse
- `Major page faults`: fallos de página que requieren acceder al almacenamiento permanente
- `Voluntary context switches`: cuando acaba el programa o al tener que esperar una operación de E/S cede la CPU a otro proceso

- `Involuntary context switches`: cuando expira su *time slice*

No es un *profiler* ya que no nos aporta mucha información interesante.

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 la otra.

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

- Instrumentación en la compilación opción `-pg`:

  ```
  gcc prog.c -o prog -pg -g
  ```

- Ejecución del programa y recogida de información:

  ```
  ./prog
  ```

  La información recogida se deja en el fichero `gmon.out`

- Visualización de la información referida a la ejecución del programa

  ```
  gprof prog
  ```

¿Cómo funciona un programa instrumentado por `gprof`? MUY IMPORTANTE

Arranque del programa

- Genera una tabla con la dirección física en memoria de cada función del programa. Se incluye al de las funciones de las bibliotecas con las que se enlaza el programa
- Se inicializan contadores de **cada función** del programa `a.o`. Hay dos contadores por función: c_1 para medir el número de veces que se ejecuta y c_2 para estimar su tiempo de CPU
- El SO programa un temporizador (por defecto $0,01s$) que se decrementará cada vez que se ejecute código del programa

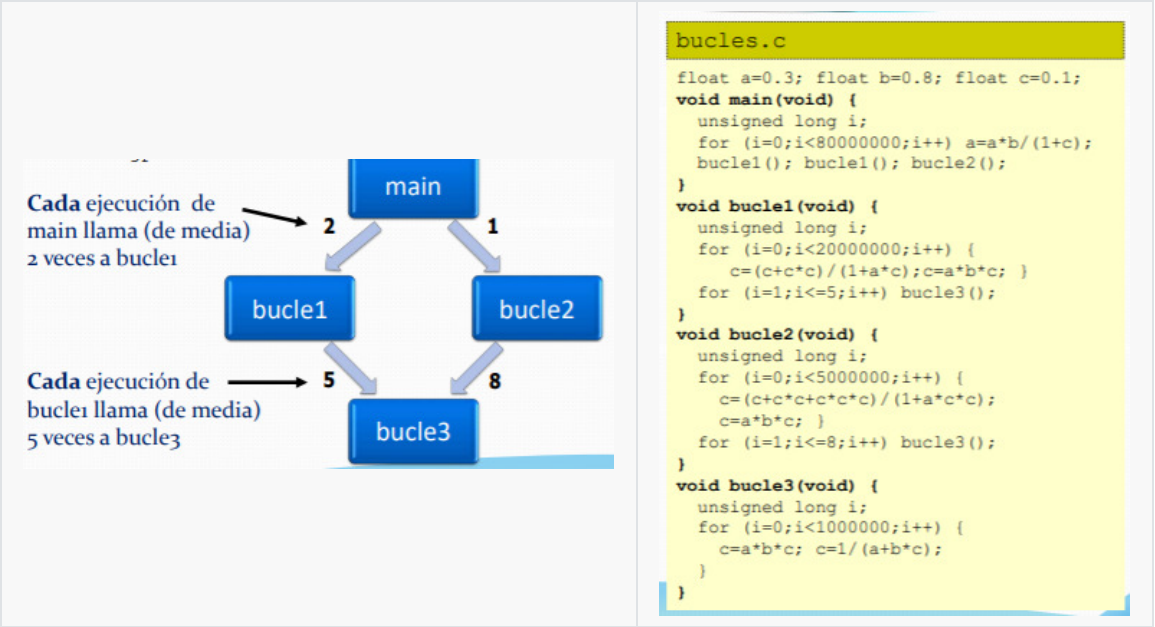
Durante la ejecución del programa

- Cada vez que se ejecuta una función se incrementa el contador c_1 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 c_2 de la función interrumpida. Se reinicia el temporizador

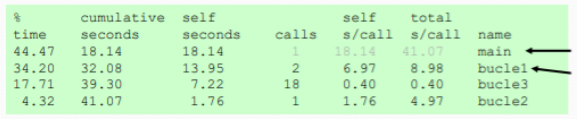
Al terminar el programa

- Teniendo en cuenta el tiempo total de CPU del programa y los contadores c_2, se **estima** el tiempo de CPU de cada función
- Se generan el *flat profile* (la tabla "básica" de un *profiler*) y el *call profile* a partir de la información recopilada

Ejemplo `gprof`



flat profile



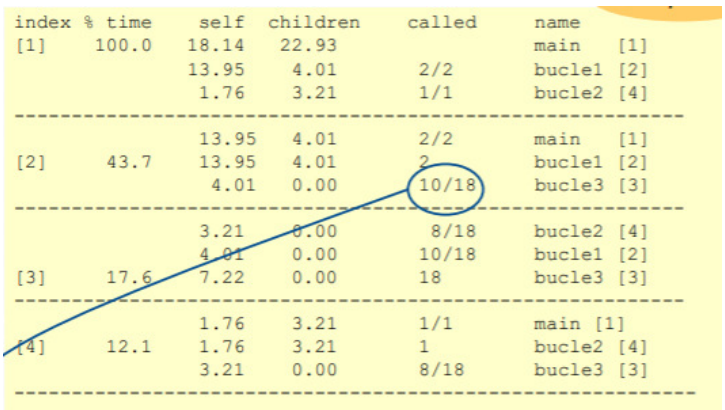
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
44.47	18.14	18.14	1	18.14	41.07	main
34.20	32.08	13.95	2	6.97	8.98	bucle1
17.71	39.30	7.22	18	0.40	0.40	bucle3
4.32	41.07	1.76	1	1.76	4.97	bucle2

- `% 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 por la subrutina y por las subrutinas que aparecen encima de ella en la tabla
- `self seconds`: tiempo de ejecución del código propio de la subrutina. Se ordena por este campo.

  $\texttt{self seconds}_i = \texttt{cumulative seconds}_i - \texttt{cumulative seconds}_{i-1}$

  $\texttt{self seconds} = \texttt{calls} \times \texttt{self s/call}$
- `self s/call`: tiempo medio de ejecución del código propio por cada llamada a la subrutina
- `total s/call`: tiempo medio total de ejecución por cada llamada a la subrutina, es decir, contando a las subrutinas a las que esta llama.

`call profile`



```
index % time    self  children    called     name
[1]    100.0   18.14   22.93                 main   [1]
                13.95    4.01       2/2       bucle1 [2]
                 1.76    3.21       1/1       bucle2 [4]
-----------------------------------------------------------
                13.95    4.01       2/2       main   [1]
[2]     43.7    13.95    4.01       2         bucle1 [2]
                 4.01    0.00      10/18      bucle3 [3]
-----------------------------------------------------------
                 3.21    0.00       8/18      bucle2 [4]
                 4.01    0.00      10/18      bucle1 [2]
[3]     17.6    7.22     0.00      18         bucle3 [3]
-----------------------------------------------------------
                 1.76    3.21       1/1       main [1]
[4]     12.1    1.76     3.21       1         bucle2 [4]
                 3.21    0.00       8/18      bucle3 [3]
-----------------------------------------------------------
```

Miramos el campo `index` y en `name` lo localizamos:

- Por encima: Las funciones que lo llaman
- Por debajo: Las funciones a las que llama

Ejercicio 3.16

El resultado de la monitorización de la actividad de una aplicación informática que está siendo ejecutada dentro de un servidor dedicado a streaming de vídeo se muestra a continuación (nótese que hay información no disponible).

Como información adicional, el perfil de llamadas indica que todos los procedimientos son llamados únicamente desde el programa principal `main` (que solo se ejecuta una vez y cuyo tiempo propio de ejecución se puede despreciar), excepto `ordena`, que solo es llamado desde el procedimiento `procesa`.

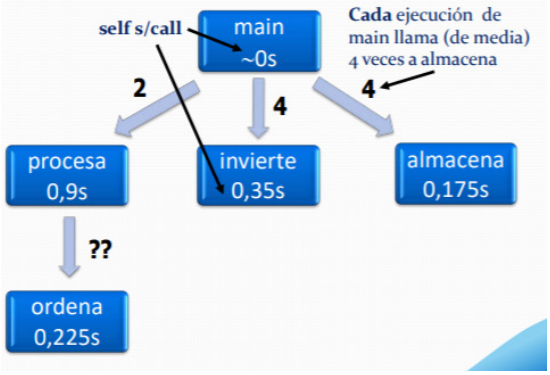
Apartado a)

Complete la información no disponible en la tabla. ¿Cuánto tiempo de CPU consume la aplicación?

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
		1.8		225	225	ordena
			2	900		procesa
		1.4	4	350	350	invierte
			4	175		almacena

Paso 1: Grafo de Llamadas

... todos los procedimientos son llamados únicamente desde el programa principal `main` (que solo se ejecuta una vez y cuyo tiempo propio de ejecución se puede despreciar), excepto `ordena`, que solo es llamado desde el procedimiento `procesa`



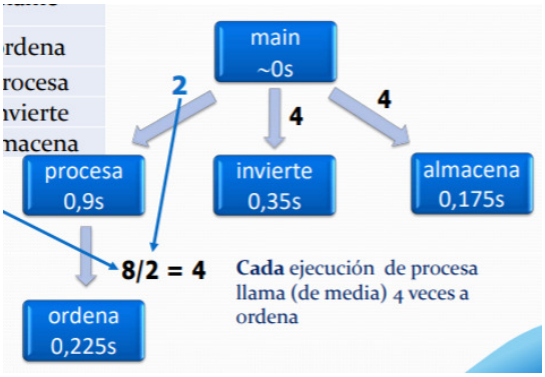
Paso 2: Empezamos a rellenar la tabla

$$self\ seconds = calls * \frac{self\ ms/call}{1000,s/s} \text{ y } calls = \frac{self\ seconds}{\frac{self\ ms/call}{1000ms/s}}$$

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
xx	xx	1,8	1,8/0,225 = 8	225	225	ordena
xx	xx	1,8	2	900	xx	procesa
xx	xx	1,4	4	350	350	invierte
xx	xx	0,7	4	175	xx	almacena

Paso 3: Completar el grafo de llamadas

Sin embargo hay que tener en cuenta que las `calls` de `ordena` son "totales", ya que `procesa` es llamado dos veces por `main`. Por lo que el grafo quedaría:



Paso 4: Rellenar `cumulative`

Sumando las filas anteriores:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
xx	1,8	1,8	8	225	225	ordena
xx	1,8+1,8 = 3,6	1,8	2	900	xx	procesa
xx	1,8+1,8+1,4 = 5,0	1,4	4	350	350	invierte
xx	1,8+1,8+1,4+0,7 = 5,7	0,7	4	175	xx	almacena

Vemos que la aplicación consume $5,7s$ de CPU (570 muestras)

Paso 5: `% time`

$$\% \; time = 100 \times \frac{self \; seconds}{CPU \; time}$$

Paso 6: `total ms/call`

$$total \; ms/call(P) = self \; ms/call + \sum_{i=0}^{n}(num\_llamada_i * total \; ms/call(P_i))$$

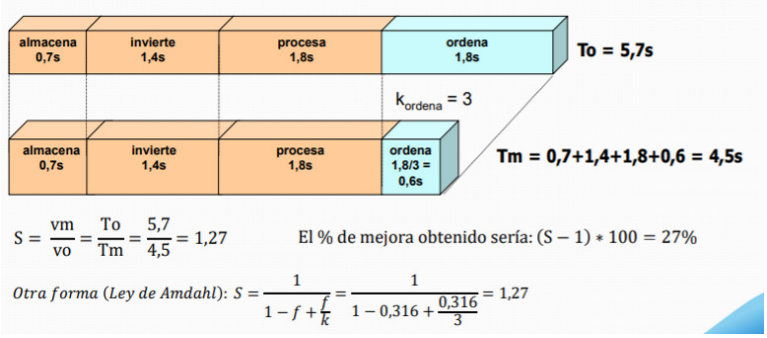
Siendo P_i cada subrutina que P llama $num\_llamada_i$ veces en su código. En este ejemplo:

cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
1,8	1,8	8	225	225	ordena
3,6	1,8	2	900	1800	procesa
5,0	1,4	4	350	350	invierte
5,7	0,7	4	175	175	almacena

$$total \; ms/call(procesa) = self \; ms/call(procesa) + 4 * total \; ms/call(ordena) = 900ms + 4 * 225ms = 1800ms$$

Apartado b)

Determine la ganancia en velocidad (speedup) que se obtendría si reemplazamos el procedimiento ordena por otro 3 veces más rápido. Exprese esa ganancia en velocidad también como tanto por ciento de mejora.



Monitor `gcov`

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

- Compilar con `-fprofile-arcs -test-coverage`
- Ejecutar
- Visualizar con `gcov prog.c` (genera `prog.c.gcov`)

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: `list, stat, record, report, annotate`

`Valgrind`

Conjunto de herramientas para el análisis y mejora del código. Encontramos:

- `Callgrind`: versión refinada de `gprof`
- `Cachegrind`: profiler de caché
- `Memcheck`: detector de errores de memoria

`Valgrind` puede analizar cualquier binario ya compilado (no necesita instrumentar). Actúa como una máquina virtual que emula la ejecución de un ejecutable en un entorno aislado

Como desventaja, el sobrecoste computacional es muy alto, puede tardar decenas de veces más que la ejecución directa del programa

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

Hacen uso de eventos software y hardware. Funcionan en Windows y en Linux.

También se pueden usar como depuradores, permiten la ejecución remota y son capaces de medir también el rendimiento de GPU, controlador de memoria...