Mejorando la Performance en Sistemas Linux/Unix

Diego Bravo E.

9 de Octubre de 2005

Tabla de contenidos

Se proporciona una metodología práctica para mejorar la performance de sistemas Linux, la cual puede ser aplicable a variantes Unix.

Cualquier sugestión o corrección, favor escribir a *diego_bravo_estrada at yahoo dot com*[1].

1. Introducción

Este es un texto breve con algunas sugestiones acerca de cómo mejorar la performance de un sistema Linux (o Unix.) No pretende cubrir el material que se detalla en los libros correspondientes, sino tan solo ser una primera aproximación para quien tiene interés en el particular y desea una explicación rápida.

Se propone la siguiente metodología secuencial para mejorar la performance:

1. Eliminar los factores negativos que deterioran la performance (higiene)
2. Analizar y reconfigurar la memoria de ser necesario
3. Analizar y reconfigurar la relación %CPU vs %I/O

1.1. Autoría y Copyright

2. Higiene

Asumiendo que nuestro sistema puede mejorar su performance (en la mayoría ocurre así), lo primero es corregir ciertos problemas típicos. En particular:

- Eliminar procesos "pesados" inútiles
- Re-escribir procesos "pesados" útiles

Llamaremos "pesado" a un proceso que torna lento al sistema. Esto ocurre comunmente porque aquél consume muchos recursos de procesamiento (mucho CPU) o consume mucha memoria.

2.1. Consumo excesivo de CPU de un proceso

Muchas veces un proceso que está fuera de control (por un error en su programación) entra en loops ilimitados que consumen inutilmente el CPU; en otros casos, esto ocurre de manera "normal" durante la ejecución de un proceso frecuente[4]. Más allá de analizar el por qué ocurre esto (que es más responsabilidad del programador), preocupémonos por detectarlo. Para esto, la manera más sencilla quizá sea emplear el comando:

```
# ps axu
```

(Consúltese el manual de `ps` para una explicación de sus opciones.) En particular, estamos interesados en la columna que reza "%CPU". Esta columna proporciona

el porcentaje que representa el tiempo de CPU consumido por el proceso respecto al tiempo total de su ejecución.

El siguiente programa se ejecuta durante 20 segundos. Los 10 primeros el proceso no consume CPU, y luego inicia un loop sin descanso hasta que transcurran otros 10 segundos. Más abajo se muestra el monitoreo del mismo en diversos momentos para un sistema sin carga:

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>

#define DELAY 10

main()
{
volatile int z;
time_t t;

/* dormir DELAY segundos */
sleep(DELAY);

/* loop infinito durante DELAY segundos */
t=time(NULL);
for(z=0;;z++)
if(time(NULL)>=t+DELAY)
break;
return 0;
}
```

Ejemplo de uso[5]:

```
$ ./sleep_and_run &
[1] 1194
$ ps axu|head -1
USER   PID  %CPU %MEM   VSZ  RSS STAT START  TIME COMMAND
$ ps axu|grep sleep_and_ru[n]
diego  1141  0.0  0.0  1376  252 S    22:01  0:00 ./sleep_and_run
$ ps axu|grep sleep_and_ru[n]
diego  1141  0.0  0.0  1376  252 S    22:01  0:00 ./sleep_and_run
$ ps axu|grep sleep_and_ru[n]
diego  1141 31.6  0.0  1376  256 R    22:01  0:02 ./sleep_and_run
$ ps axu|grep sleep_and_ru[n]
diego  1141 49.8  0.0  1376  256 R    22:02  0:05 ./sleep_and_run
$ ps axu|grep sleep_and_ru[n]
diego  1141 51.2  0.0  1376  256 R    22:01  0:06 ./sleep_and_run
$ ps axu|grep sleep_and_ru[n]
$
```

Como se aprecia, la columna `%CPU` empieza a crecer tras unos momentos hasta alcanzar un valor significativo (51.2%.)

Lamentablemente, este procedimiento no detecta los procesos que de pronto salen de control si éstos ya se han venido ejecutando durante mucho tiempo manteniendo un consumo moderado de CPU (pues en ese caso el `%CPU` tardará mucho en hacerse significativo.)[6]

Afortunadamente, existe otra forma muy sencilla de analizar este caso, y consiste en verificar la columna `TIME` del proceso en diversos momentos. Si se aprecia el ejemplo anterior, se observa que la columna `TIME` se mantiene estable (en este caso, con `0:00`) durante los primeros momentos, y luego rápidamente se incrementa. Esto es un indicativo claro de que el proceso está consumiendo mucho de nuestro CPU.

En este punto, el administrador deberá matar inmediatamente al proceso, si éste resulta ser inútil. En cambio, si el proceso cumple alguna función de utilidad en el sistema, se deberá esperar un tiempo prudencial para que vuelva a la normalidad (deje de consumir tanto CPU) o para que termine.

2.2. Consumo excesivo de Memoria de un proceso

A veces asociado con lo anterior, los procesos pueden consumir demasiada memoria por distintos motivos:

- Errores de programación, especialmente al no liberar los objetos
- La memoria física es insuficiente para la cantidad de información que se requiere utilizar

Una manera rápida de saber si el sistema en su conjunto tiene carencia de memoria física consiste en analizar las columnas '`si`' y '`so`' bajo '`--swap--`'[7]:

```
$ vmstat 2
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo
 0  0      0 321408  11960  95300    0    0    36     9
 0  0      0 321424  11960  95300    0    0     0     4
 0  0      0 321424  11960  95300    0    0     0     0
```

Si los valores en las columnas mencionadas son cero (quizá con pequeñas excepciones esporádicas), quiere decir que el sistema tiene suficiente memoria física para los requerimientos DE ESTE MOMENTO. Por lo tanto, el administrador hará bien en analizar la salidad de `vmstat` en diferentes momentos del día, o cuando se ejecutan los procesos cuya lentitud nos preocupa más.

Con el fin de analizar la memoria consumida por los procesos, podemos referirnos a las columnas `%MEM` y `RSS` de la salida nuestro conocido '`ps axu`', que son equivalentes. `RSS` corresponde a la memoria que el sistema operativo ha otorgado para el proceso, el cual se presenta como porcentaje respecto al total en `%MEM`.

Siguiendo la idea, es una buena idea analizar qué procesos consumen mucha memoria y desactivarlos si no son imprescindibles. Más abajo se explica con detalle qué se puede hacer si tenemos un sistema con carencia de memoria en el que no podemos desactivar los procesos culpables.

2.3. Re-escritura de procesos pesados

No pretendo entrar al tópico de optimizar la ejecución de los programas. En mi opinión, los buenos programadores casi siempre pueden optimizar la ejecución de sus programas para que consuman menos CPU o Memoria, aunque no es una tarea trivial que conlleve poco tiempo. El problema aquí no es técnico, sino de "gerencia": el programador suele ser obligado a liberar una versión operativa que rápidamente pasa a producción, y luego se le reasigna a otra actividad sin darle tiempo para (ni obligarle a) que la optimice. Los jefes de IT se sorprenderían gratamente si asignaran a sus buenos programadores exclusivamente a optimizar las aplicaciones por algunos días.

Un aspecto imprescindible de resaltar porque resulta evidente, es el consumo excesivo de recursos que hacen los programas desarrollados en lenguajes tipo scripting como el Shell. Si bien es cierto que programar en el Shell (o similares) tiene múltiples ventajas, se debe tener en cuenta que una aplicación equivalente reescrita en lenguaje C suele ser de 10 a mil veces más veloz, y por tanto mucho menos derrochadora de CPU y Memoria.

El shell tiene su lugar para diversas tareas administrativas y para prototipado de cierta clase de aplicaciones, pero rara vez debería constituirse en el lenguaje de una aplicación principal. Como se sabe, muchas veces ocurre que la falta de tiempo o la desidia hace que la gente utilice el prototipo y postergue (para siempre) el desarrollo "en serio".

3. Configuración de la Memoria RAM

Como se indicó en la introducción, una vez que se ha aplicado las normas de "higiene" en el sistema, tenemos dos posibilidades para mejorar aún más su performance: la configuración de la memoria y la repartición CPU-I/O. Aunque ambos análisis se pueden hacer en paralelo, es muy conveniente hacerlos secuencialmente a fin de cuantificar separadamente el impacto y simplificar las observaciones. Asimismo, es bastante recomendable empezar con la configuración de la memoria, dado que las soluciones por este concepto son más sencillas y menos costosas que en lo concerniente a la repartición CPU-I/O.

3.1. Sistemas sin carencia de memoria

Algunos sistemas nunca tienen absolutamente ninguna carencia de memoria. Una forma sencilla de determinar esta situación es observando si han hecho uso del swap. Como se aprecia en el ejemplo, el swap (en este caso de aproximadamente 1200 megabytes) no tiene uso HASTA ESTE MOMENTO, lo que significa que el sistema todavía no ha tenido que recurrir al él por falta de memoria:

```
sys1$ /sbin/swapon -s
Filename            Type            Size    Used    Priority
/dev/hda7           partition       1228932 0       -1
```

Si esta situación se mantiene a lo largo de muchos días, es indicativo de que dicho sistema probablemente tiene exceso de memoria, o cuando menos, exceso de swap. En estos sistemas, dificilmente podríamos mejorar la performance en lo referente al consumo de memoria.

3.2. Régimen de carencia de memoria

Normalmente los sistemas experimientan carencias de memoria durante ciertos lapsos en los que se ejecutan determinados procesos pesados. Si estos "períodos de carencia" son breves, entonces no hay mucho que mejorar en cuanto a la falta de memoria. Por el contrario, si los períodos de carencia son prolongados, la performance se puede multiplicar asombrosamente.

En conclusión, el primer paso consiste en determinar si nuestro sistema tiene largos períodos de carencia de memoria (por lo menos de algunos minutos) o si éstos son breves y poco numerosos (de pocos segundos de duración.) Para el primer caso se aconseja una reconfiguración, mientras que para el segundo una reconfiguración mejoraría muy poco la performance del conjunto y probablemente no valdría el esfuerzo.

3.3. Observación de la carencia de memoria

A continuación se muestra parte de la salida de `vmstat` para un sistema que inicialmente no tiene carencias de memoria, pero luego de unos segundos sí que las tiene:

```
$ vmstat 2
---------memory- ---swap-- -----io---- --system-- ----cpu----
  swpd   free     si   so    bi    bo   in    cs us sy id wa
 67288 478472    188  262   282   279 1037   239  8  1 81 10
 67288 478472      0    0     0     0 1002    59  2  0 99  0
 67288 478472      0    0     0     0 1001    54  2  0 99  0
 67288 478472      0    0     0    24 1010   103  1  0 99  0
 67288 478456      0    0     0     0 1017   467  8  1 91  0
 67236 102320     34    0    46    16 1023   297 11 54 35  2
 72148   3336   1276 4202  2446  4204 1424   459 15 23  0 61
111640   2960   1674 20072  4210 20086 1263   273  1  6  0 93
129996   3208   1960 9180  3622  9180 1190   327  1  4  0 96
129996   3568   3200    0  3862     0 1245   480  1  2  0 97
129996   3176   2454    0  2984     6 1225   414  4  2  0 95
129988   3496   3672    0  4076    68 1328   567  2  2  0 96
```

```
129988   2892   3268    0  4396     2 1325   531  0  2  0 97
175224   2768   2864 22664  3186 22710 1309   455  1  7  0 92
175224   3016   2460    0  3030     0 1254   675  5  3  0 92
 76508 474376    438    0   478     0 1060   189  2 10 70 20
 76508 474388      6    0     6     2 1012   436  8  1 91  1
 76508 474404      0    0     0     0 1001    54  2  0 99  0
 76508 474408      0    0     0     0 1001    57  1  0 99  0
 76508 474408      0    0     0     0 1001    51  2  0 99  0
```

Obviando la primera línea (que son valores acumulados), apreciamos que bajo las columnas 'si' y 'so' (swap in/swap out) los valores permanecen inicialmente en cero, y luego 'so' se dispara; luego de unos instantes 'si' también adquiere valores significativos. Finalmente tras unos 20 segundos, ambas columnas retornan a cero.

Este es un ejemplo típico de un sistema que durante unos segundos experimienta carencia de memoria. Los usuarios de este sistema probablemente experimentarán lentitud en la respuesta durante estos momentos.

¿Qué debemos hacer al respecto? si durante todo el día éste es el único momento de carencia de memoria, entonces cualquier reconfiguración de memoria sólo nos permitiría ganar hasta 20 segundos en todo el mismo día, lo cual es evidentemente despreciable. Si por el contrario, este comportamiento hubiera durado algunas horas, o si ocurriera en muchas ocasiones (que acumulando su duración podría totalizar horas) entonces la reconfiguración sí es aplicable.

3.4. Reconfiguración de la memoria

Si hemos decidido reconfigurar la memoria a partir del análisis anterior, entonces tenemos esencialmente tres opciones:

1. Modificar el orden de ejecución de los procesos
2. Modificar los programas
3. Ampliar la memoria

Describiremos cada caso por separado.

3.4.1. Modificar el orden de ejecución de los procesos

Este es el primer nivel de solución al problema de la carencia de memoria; es el más sencillo y el que requiere menos inversión.

A modo de ejemplo, considérese un sistema en el cual dos procesos que se ejecutan separadamente uno detrás del otro tardan (en total) cinco minutos en completarse; pero si se ejecutan simultáneamente tardan una hora o más en terminar. Esto ocurre así debido a que cada proceso por separado no introduce al sistema en el régimen de carencia de memoria o lo hace por muy poco tiempo e intensidad, mientras que la combinación de ambos procesos sí lo hace de lleno y la performance se reduce drásticamente.

Este problema ocurre típicamente en sistemas en los que el número de procesos es muy elevado y por simplicidad para los operadores, se prefiere la ejecución simultánea. Evidentemente, la solución consiste en identificar las "combinaciones" que generan el régimen de carencia de memoria y serializar la ejecución.

3.4.2. Modificar los programas

Las consideraciones acerca de la optimización que se comentaron en la sección de "higiene" son totalmente aplicables aquí. Lamentablemente muchas instalaciones ejecutan programas de terceros o no cuentan con el personal adecuado para este trabajo, por lo que sólo pueden recurrir a la ampliación de la memoria física (ver más abajo.)

En muchos casos (especialmente programas no documentados) este procedimiento puede resultar costoso (en tiempo de honorarios de programación), poco seguro (pues tal vez el programa ya no se puede optimizar mucho en cuanto a su consumo de memoria), y lento (pues la optimización puede resultar complicada y requiere muchas horas de análisis) por lo que también se recurre a la ampliación de la memoria física.

3.4.3. Ampliar la memoria física

En este caso sólo hay que determinar cuánta memoria se va a agregar. En la mayoría de casos la gente opta por agregar "bancos" de memoria que suelen duplicar la memoria existente, y volver a efectuar el análisis de la performance para cerciorarse de que ya no hay carencia de aquella.

Esta aproximación iterativa es válida, pero en algunos casos es aconsejable tener una idea más exacta (por ejemplo, si el dinero escasea, o si no hay disponibilidad de "bancos" de memoria en el mercado, o si nuestro hardware está casi al límite de su capacidad de crecimiento en memoria.)

Durante la operación del sistema es frecuente que se observen diversos episodios de carencia de memoria de poca importancia, pero que necesariamente irán utilizando nuestro swap. Si volvemos a la salida de `vmstat` del ejemplo anterior, apreciaremos que antes de que se inicie el período crítico el consumo de swap era de 67288 bloques (también se pudo obtener con `swapon -s`.) Este consumo se eleva hasta 175224 bloques para luego disminuir gradualmente. Esto quiere decir que el proceso pesado excedió en 175224-67288=107936 bloques la capacidad de nuestra memoria empleando el swap. Por tanto, si agregamos precísamente esta cantidad de memoria física (por ejemplo, un "banco" de 128Mb) el proceso pesado podrá ejecutarse enteramente en RAM[8].

Evidentemente este tipo de análisis requiere bastante paciencia y los resultados pueden ser variables de un día para otro en función de la carga total del sistema.

4. Análisis CPU-I/O

Pregunta motivadora: Si dos procesos toman normalmente dos horas en ejecutarse, y un buen día adquirimos un CPU (quizá con la memoria respectiva) que proporciona el DOBLE de velocidad de procesamiento, ¿cuánto tardarán ahora estos procesos?

La respuesta, evidentemente, NO es una hora.

4.1. Recursos consumidos por los procesos

Los procesos que se ejecutan en los computadores hacen uso de diversos subsistemas del computador, tales como el CPU, la Memoria, los discos, y quizá la red. Normalmente se asume que la memoria tiene una velocidad "compatible" con la del CPU[9], por lo que los tiempos consumidos en ambos se combinan en uno sólo: el "tiempo de CPU".

Si un proceso no está ejecutando instrucciones del CPU en un momento dado, normalmente es porque está accediendo al disco o a la red (coloquialmente, se dice que está efectuando I/O.) Existe una tercera posibilidad que corresponde a procesos que "pausan" o "esperan eventos" sin hacer nada más (estado "idle" u "ocioso".)

4.2. Repartición CPU-I/O-Idle

A partir de lo anterior, para cualquier proceso se puede plantear que el tiempo de su ejecución se divide en tres componentes:

```
Tiempo Total = Tiempo CPU + Tiempo I/O + Tiempo Idle
```

Esta relación se puede acumular a todos los procesos del sistema y expresarla porcentualmente para un lapso de tiempo determinado:

```
% CPU sistema + % I/O sistema + % Idle sistema = 100%
```

El comando `vmstat` en sus últimas columnas nos proporciona estos valores. Por ejemplo:

```
$ vmstat 5
procs ------ ---swap-- -----io---- --system-- ----cpu----
 r  b  cache   si   so    bi    bo   in    cs us sy id wa
 2  0 110144    0    0    20     7 1013   237  4  1 94  1
 0  1 120520    0    0  2166    21 1160  1424 12  4 40 45
 6  0 133916    0    0  2573   133 1168  1311 31  4 27 38
 0  1 201372    0    0 13646   197 1155  1227 18  6 22 53
 0  1 210380    0    0  1870   201 1096  1305 62  4  0 33
 0  0 212548    0    0   437   163 1162  1049 16  2 77  5
```

La primera fila generada por `vmstat` debe ser descartada para este análisis pues corresponde a un acumulado desde que el sistema fue encendido.

Nuestro `%CPU` del sistema corresponde a la suma de las columnas '`us`' y '`sy`' (que se comentan luego.) El `%I/O` se encuentra en la columna '`wa`', y por último, el `%Idle` corresponde a la columna '`id`'.

NOTA: El manual de `vmstat` indica que para kernels Linux de versión menor a 2.5.41, la columna '`wa`' siempre reporta cero, y la columna '`id`' contiene la suma de `%Idle+%I/O`[10].

El comando `vmstat` acepta, entre otros, un argumento numérico que corresponde al intervalo de muestreo (en segundos.) Si deseamos conocer la distribución de los tiempos del sistema durante la ejecución de ciertos procesos, debemos lanzar `vmstat` en paralelo con los mismos y promediar los valores obtenidos mientras éstos se ejecutan. A modo de ilustración, asumiremos que pretendo analizar la distribución de tiempos de un proceso que tarda aproximadamente dos horas; `vmstat` se lanza con intervalo de muestreo de 20 minutos (1200 segundos):

```
$ vmstat 1200
procs ------- ---swap-- -----io---- --system-- ----cpu----
 r  b   cache   si   so    bi    bo   in    cs us sy id wa
 2  0  212640    0    0    31     7 1013   235  4  1 94  1
 2  0  220324    0    0  3844   296 1021  2360 82 10  0  9
 0  0  223900    0    0  1800     0 1087  1692 38  4 45 13
 4  0  224144    0    0     0     0 1205  3511 74 12 13  0
 1  0  226120    0    0   972   950 1223  2169 34  6 26 33
 1  0  229008    0    0  1490     0 1070  2471 52  8 19 21
 0  0  229916    0    0   564   320 1243  3014 52 10  6 31
```

Descartando la primera fila, los tiempos obtenidos son:

Tabla 1.

%CPU	%I/O	%IDLE	%TOTAL
92	9	0	101
42	13	45	100
86	0	13	99
40	33	26	99
60	21	19	100
62	31	6	99

No es necesario preocuparse porque el total no suma exactamente 100%. A continuación el promedio de las columnas relevantes[11]:

Tabla 2.

%CPU	%I/O	%IDLE
63.7	17.8	18.2

Esta distribución es extremadamente útil para averiguar en dónde está concentrada la "lentitud". En este ejemplo, claramente se aprecia que el %CPU es el factor principal, pero tanto el %I/O como el %Idle son significativos. Por lo tanto, el diagnóstico es: Mejorar todos los subsistemas, pero especialmente el CPU.

Asumiendo que el proceso ha tomado dos horas (120 minutos) en total, de acuerdo a la tabla anterior los tiempos absolutos han sido:

Tabla 3.

T. CPU	T. I/O	T. IDLE
76.4 min	21.4 min	21.8 min

4.2.1. La pregunta inicial

Con los valores de la última tabla, podemos responder a la pregunta con que se inició esta sección: ¿Cuánto tardarán en total los procesos si el CPU es el doble de veloz?

Evidentemente, si el CPU es el doble de veloz, el tiempo en que éste es empleado se reducirá a la mitad. Para nuestro caso:

```
T. CPU = 76.4 min / 2 = 38.2 min
```

Con lo que nuestros tiempos resultan siendo:

Tabla 4.

T. CPU	T. I/O	T. IDLE
38.2 min	21.4 min	21.8 min

Finalmente, el tiempo total de los procesos será:

```
T. Total = 38.2 min + 21.4 min + 21.8 min = 1h 21min
```

En general, si el tiempo normal que tardan los procesos es `To`, y el CPU mejora en un factor `F` (en nuestro ejemplo, `F=2`), entonces el nuevo tiempo total `Tn` será:

```
Tn = To [1-(%CPU/100).(1-1/F)]
```

Para nuestro ejemplo:

```
Tn = 120 [1-(63.7/100).(1-1/2)] = 81.78 min = 1h 21min
```

Evidentemente este mismo análisis se puede aplicar para el subsistema de I/O, correspondiendo al incremento (o disminución) de la velocidad de los discos, o en ocasiones al incremento de la memoria física que permite el crecimiento del caché de disco.

Finalmente, téngase en cuenta que en Linux, el tiempo de espera con la red se considera tiempo Idle.

4.2.2. User Time y System Time

Anteriormente (en la salida de `vmstat`) se explicó que el %CPU estaba constituido por la adición de los valores 'us' (User Time) y 'sy' (System Time.) Estos valores pueden ser útiles cuando se pretende optimizar el código de un programa a fin de que consuma menos tiempo de CPU.

El "User Time" corresponde al tiempo que el CPU invierte en ejecutar el texto de los programas en cuestión (cálculos, toma de decisiones, acceso a memoria), mientras que el "System Time" es el tiempo que el CPU invierte en ejecutar código **del kernel** en función de las solicitudes que el programa le hace (system calls.)

La observación de estos valores puede hacerse para un proceso particular (pues `vmstat` obtiene el promedio de todo el sistema) mediante el comando `time`. Por ejemplo:

```
$ time find /etc > /dev/null
find: /etc/rcS.d: Permission denied
find: /etc/ssl/private: Permission denied
find: /etc/amanda: Permission denied

real    0m4.077s
user    0m0.003s
sys     0m0.050s
$
$ time find /etc > /dev/null
find: /etc/rcS.d: Permission denied
find: /etc/ssl/private: Permission denied
find: /etc/amanda: Permission denied

real    0m0.047s
user    0m0.006s
sys     0m0.018s
```

Este ejemplo muestra que el comando especificado (`find`) en su primera ejecución tardó en total 4.077 segundos en completarse. Durante este tiempo, sólo tres milésimas correspondieron a la ejecución del código del programa (pues `find` no hace mucho procesamiento de información), y cincuenta milésimas fueron dedicadas al código del kernel (en el caso de `find`, este código del kernel se encarga de navegar en una estructura de directorios.) Nótese que el tiempo total de CPU corresponde a sólo 53 milésimas: muy lejos de los cuatro segundos totales. Esto se debe a 1) otros procesos también se están ejecutando, y 2) el proceso tiene que esperar a que el disco entregue la información solicitada, lo cual es varios órdenes de magnitud más lento que las operaciones de CPU.

En la segunda ejecución el tiempo total se reduce radicalmente a 47 milésimas (ochenta y seis veces más rápido), lo cual se debe sin duda a que los datos de la primera ejecución están ahora en el caché de disco y ya no hay necesidad de ir al disco físico. Nótese que los tiempos de CPU también han cambiado, quizá por el efecto de otros procesos simultáneos, y (para el caso del Kernel/System Time) porque el mecanismo de acceso a la información de la memoria caché resulta más sencillo que el de un acceso real al disco físico[12].

El conocimiento de esto nos puede dar pistas con respecto a la estrategia de optimización de un proceso pesado. Por ejemplo, para el caso anterior, es evidente que no sirve de mucho tratar de optimizar el (escazo) procesamiento que lleva a cabo `find`, sino enfocar los esfuerzos a emplear menos tiempo en el Kernel. Si bien no trataremos de reprogramar el Kernel, al menos sí podemos tratar de reducir las llamadas que hacemos al mismo desde nuestro programa, o buscar otras llamadas más eficientes.

A modo de ejemplo, el siguiente programa ejecuta un cálculo (inútil) durante cinco segundos, y reporta cuantos millones de iteraciones se ha realizado dicho cálculo:

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

int main(void)
```

```
{
int x=1, y=2, z=3;
long iter1=0,iter2=0;
struct timeval tv1,tv2;

gettimeofday(&tv1,NULL);
for(;;)
{
x=(x*3+y*7+z*9)%11;
y=(x*9+y*11+z*3)%29;
z=(x*17+y*13+z*11)%37;
iter1++;
if(iter1==1000000)
{
iter2++;
iter1=0;
}
gettimeofday(&tv2,NULL);
if(tv2.tv_sec==tv1.tv_sec+5 && tv2.tv_usec>=tv1.tv_usec ||
tv2.tv_sec>tv1.tv_sec+5)
break;
}
printf("Iteraciones: %ldM Resultado: %d %d %d\n",iter2,x,y,z);
return 0;
}
```

Por ejemplo, en mi computador obtuve:

```
$ time ./calculo1
Iteraciones: 12M Resultado: 6 4 5

real    0m5.002s
user    0m2.379s
sys     0m2.461s
```

Claramente se aprecia que el "system time" es similar al "cpu time". Los 2.379 segundos de "user time" sólo alcanzaron para 12 millones de iteraciones.

Este programa tiene evidentemente el inconveniente de obtener la hora en cada iteración, lo que deriva en múltiples solicitudes al kernel. Esto se puede mejorar, por ejemplo, preguntando la hora cada millón de iteraciones:

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

int main(void)
{
int x=1, y=2, z=3;
long iter1=0,iter2=0;
struct timeval tv1,tv2;

gettimeofday(&tv1,NULL);
for(;;)
{
x=(x*3+y*7+z*9)%11;
y=(x*9+y*11+z*3)%29;
z=(x*17+y*13+z*11)%37;
iter1++;
if(iter1==1000000)
{
iter2++;
iter1=0;
gettimeofday(&tv2,NULL);
if(tv2.tv_sec==tv1.tv_sec+5 && tv2.tv_usec>=tv1.tv_usec ||
   tv2.tv_sec>tv1.tv_sec+5)
break;
}
}
printf("Iteraciones: %ldM Resultado: %d %d %d\n",iter2,x,y,z);
return 0;
```

```
}
```

Con esto, el tiempo del proceso ha sido casi exclusivamente "User Time", lo cual alcanzó para 65 millones de iteraciones:

```
$ time ./calculo2
Iteraciones: 65M Resultado: 1 23 5

real    0m5.055s
user    0m5.030s
sys     0m0.001s
```

El único inconveniente de esta implementación es que se pierde cierta precisión en el tiempo de corte (se pasó por 55 milésimas) debido a que no verifica el tiempo con mucha regularidad. Una solución que aprovecha más recursos del sistema operativo se muestra a continuación:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handler(int);

int timeout=0;

int main(void)
{
int x=1, y=2, z=3;
long iter1=0,iter2=0;

signal(SIGALRM,handler);
alarm(5);
for(;;)
{
x=(x*3+y*7+z*9)%11;
y=(x*9+y*11+z*3)%29;
z=(x*17+y*13+z*11)%37;
iter1++;
if(iter1==1000000)
{
iter2++;
iter1=0;
}
if(timeout)
break;
}
printf("Iteraciones: %ldM Resultado: %d %d %d\n",iter2,x,y,z);
return 0;
}

void handler(int s)
{
timeout=1;
}
```

El resultado es óptimo:

```
$ time ./calculo3
Iteraciones: 65M Resultado: 8 17 19

real    0m5.002s
user    0m4.977s
sys     0m0.001s
```

El último programa falla cuando se compila con máxima optimización en `gcc` (opción `-O3`.) Esto se debe a que el optimizador descubre que la variable `timeout` no es cambiada en ningún lugar de `main()` y asume (erroneamente) que nadie más la altera, por lo que genera un código que no la verifica y el loop `for()` se torna infinito. Una forma estándar de evitar esto consiste en declarar la variable

`timeout` como `volatile` a fin de que el optimizador no haga asunciones con respecto a la misma. El código de `calculo3.c` cambia exactamente en una línea:

```
$ diff calculo3.c calculo4.c
7c7
< int timeout=0;
---
> volatile int timeout=0;
```

Tras compilar con optimización, el resultado es aún mejor: ciento tres millones de iteraciones.

```
$ gcc -O3 -o calculo4 calculo4.c
$ time ./calculo4
Iteraciones: 103M Resultado: 2 23 3

real    0m5.002s
user    0m4.977s
sys     0m0.001s
```

A. Ejercitador de memoria

El siguiente programa permite ejercitar la memoria física (y el swap con los valores adecuados) con el fin de aprender a observar la paginación (con `vmstat`):

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char **argv)
{
char *ptr;
unsigned long z,k,r;
unsigned long pag,off;
unsigned char x;
unsigned long r1,r2,r3,r4;
if(argc!=3)
{
fprintf(stderr,"Use ram_stress <#KB> <#repeticiones>\n");
return 2;
}
k=atoi(argv[1]);
r=atol(argv[2]);
if(k<1 || r<1)
{
fprintf(stderr,"Se requiere valor positivo de KB y repeticiones\n");
return 1;
}
ptr=malloc(1024L*k);
if(ptr==NULL)
{
fprintf(stderr,"malloc error: falta memoria?\n");
return 1;
}
for(z=0;z<r;z++)
{
r1=rand(); r2=rand(); r3=rand(); r4=rand();
pag=labs(r1*r2)%k;
off=r3%1024;
x=r4%256;
ptr[1024*pag+off]=x;
}
free(ptr);
return 0;
}
```

Por ejemplo, para un sistema de 512Mb, el siguiente comando obliga a utilizar aproximadamente 50Mb para pruebas (el segundo argumento es el número de "pruebas" y debe ajustarse gradualmente dependiendo de la duración deseada del test.)

```
$ time ./ram_stress 50000 1200000

real    0m0.842s
user    0m0.670s
sys     0m0.151s
```

En mi sistema (Athlon XP-2000) este comando tarda menos de un segundo en completarse. En cambio, si se reserva más memoria (aún manteniendo el número de pruebas, es decir, haciendo el mismo trabajo) el tiempo se eleva a más de 1 minuto debido a que se ingresa al régimen de carencia de memoria:

```
$ time ./ram_stress 500000 1200000

real    1m9.893s
user    0m1.027s
sys     0m1.896s
```

Ud. debería hacer estas pruebas con diferentes valores para el bloque de memoria. Observará cómo la paginación se hace cada vez más pronunciada conforme solicitamos un bloque que se acerca cada vez más a la memoria física instalada.

Notas

1. Como todos mis textos docbook, éste ha sido escrito usando el script qdk disponible en `http://qdk.sourceforge.net`.
2. http://www.gnu.org/licenses/fdl.txt
3. http://www.gatogringo.com
4. Es aceptable que un proceso consuma mucho CPU por pocos segundos o minutos. Son muy escasos los procesos que consumen mucho CPU durante tiempos extendidos.
5. Para compilar el programa, usar algo como: `cc -o sleep_and_run sleep_and_run.c`
6. En algunos sistemas Unix, el `%CPU` se toma en relación a los últimos segundos de ejecución, por lo que no se presenta este inconveniente. Por otro lado, en sistemas multiprocesador el `%CPU` suele alcanzar `100/N%` por proceso o por thread, donde `N` es el número de procesadores.
7. En algunos sistemas Unix esto se separa en columnas de "paginación" tales como 'page-in' y 'page-out'.
8. De seguro al inicio entraría en el régimen de carencia de memoria dependiendo del consumo que otros procesos hagan de esta, pero tras desplazar a aquellos al swap, el proceso pesado terminaría por completo en la RAM.
9. En ocasiones, por errores de configuración de partes de hardware, se emplean memorias muy lentas que obligan al CPU a "esperarlas", ocasionando que éste opere efectivamente a menor velocidad.
10. Para averiguar qué kernel Linux se está empleando, usar el comando "`uname -r`".
11. Si se conoce aproximadamente el tiempo total de los procesos que se analizan, se puede lanzar `vmstat` indicando este tiempo a fin de obtener una única muestra que ya no requerirá de este promediado.
12. En realidad, los tiempos para este ejemplo son demasiado breves como para tomarlos como base de análisis. Normalmente se utilizan tiempos de al menos varios minutos.