



DISEÑO Y ADMINISTRACIÓN DE SISTEMAS OPERATIVOS

PRIMERA PED: MITOP

✉ David Pastor Sanz

Índice

1. Introducción	2
2. Implementación	3
2.1. Obtención de los PIDs	3
2.2. Cálculo de los porcentajes de uso de la CPU	3
2.3. Cálculo de la memoria	5
2.4. Obtención del resto de datos	6
2.5. Impresión del resultado	7
3. Ejecución de ejemplo	9
3.1. Comparando mitop con top	9
3.2. Comparando Ejercicio1 con top	10
4. Código fuente	12
5. Opinión personal y material utilizado	15
6. Bibliografía	16

1. Introducción

Para conseguir el resultado pedido por el enunciado se ha creado un script en **Bash** y se le han dado los permisos necesarios para poder ejecutarlo. Éste simula la ejecución del comando **top**, mostrando por pantalla algunos de los datos de dicho con la máxima similitud posible. Para obtener todos los requisitos pedidos se ha tenido que acceder con distintos comandos a los diferentes archivos, del sistema de ficheros **/proc**, para obtener las referencias necesarias para, o directamente sacar o mediante distintos cálculos, adquirir los valores que se han de mostrar.

Para la realización de la práctica, ejecución y comprobación de su correcto funcionamiento se ha utilizado la máquina virtual ofrecida en el curso virtual. Esta se basa en la distribución linux **Ubuntu 12.04.5 LTS** para una arquitectura **i686**. Dicha información se ha obtenido de la siguiente manera:

- Utilizando el comando **cat**, que concatena y/o muestra archivos en la salida estándar, sobre el fichero **/etc/issue** para obtener la distribución

```
$ cat /etc/issue
```

- Mediante el comando:

```
$ uname -m
```

para obtener la arquitectura

A screenshot of a terminal window with a dark background. The window title is 'sistemas@DyASO: ~'. The first command entered is 'sistemas@DyASO:~\$ cat /etc/issue', which outputs 'Ubuntu 12.04.5 LTS \n \l'. The second command entered is 'sistemas@DyASO:~\$ uname -m', which outputs 'i686'.

Figura 1: Sistema operativo y arquitectura

Se ha editado el script mediante el programa **gedit**.

No se ha utilizado ni ha sido necesario crear ningún fichero externo para realizar y ejecutar correctamente la práctica.

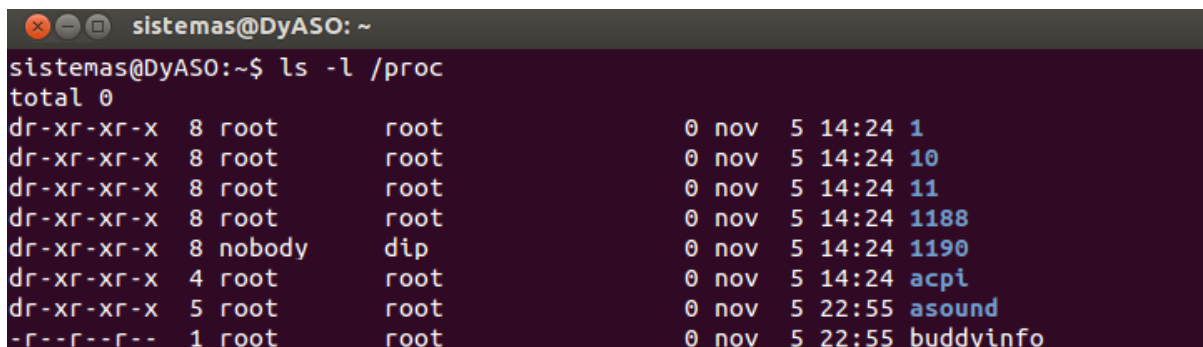
2. Implementación

Se va a proceder a explicar punto por punto los distintos datos y la manera de obtenerlos mediante el código utilizado. Además, la forma de la que el programa final muestra el resultado en el terminal al ejecutar el script. Al principio del código se declaran ciertas variables y funciones que se explicarán en el momento en el que se utilicen.

2.1. Obtención de los PIDs

Lo primero que se va a hacer es guardar todos los identificadores de los procesos (**PID**) en una variable. Para esto se va a utilizar el comando `awk`. Este comando utiliza el lenguaje de mismo nombre diseñado para procesar datos basados en texto.

Se guardarán en la variable `pids` utilizando la siguiente sentencia encerrada entre `$()`, que tiene un significado de retorno, es decir que la variable `pids` almacenará los datos que devuelva la sentencia. Para ello se llama al comando que muestre los detalles de los ficheros que se encuentran en el directorio `/proc` (esto se realiza mediante el comando `ls -l directorio`, se puede ver la información mostrada en la Figura 2). Se utiliza una tubería (`|`) para pasar los datos obtenidos al comando `awk` y que este los procese.



```
sistemas@DyASO: ~
sistemas@DyASO:~$ ls -l /proc
total 0
dr-xr-xr-x  8 root      root           0 nov  5 14:24 1
dr-xr-xr-x  8 root      root           0 nov  5 14:24 10
dr-xr-xr-x  8 root      root           0 nov  5 14:24 11
dr-xr-xr-x  8 root      root           0 nov  5 14:24 1188
dr-xr-xr-x  8 nobody    dip           0 nov  5 14:24 1190
dr-xr-xr-x  4 root      root           0 nov  5 14:24 acpi
dr-xr-xr-x  5 root      root           0 nov  5 22:55 asound
-r--r--r--  1 root      root           0 nov  5 22:55 buddyinfo
```

Figura 2: Información mostrada mediante `ls -l /proc`

La llamada a `awk` selecciona la novena columna de cada línea, es el campo en el que se encuentran los identificadores (PIDs). No todos ellos simbolizan los subdirectorios de cada proceso, solo aquellos que dicho campo son un número entero, se puede observar en la Figura 2 como por ejemplo 11 es un subdirectorio del proceso con **PID** 11, pero `acpi` no pertenece a ningún **PID**. A continuación se muestra el código utilizado:

```
pids=$(ls -l /proc | awk '$9 ~ /[0-9]/ {print $9}')
```

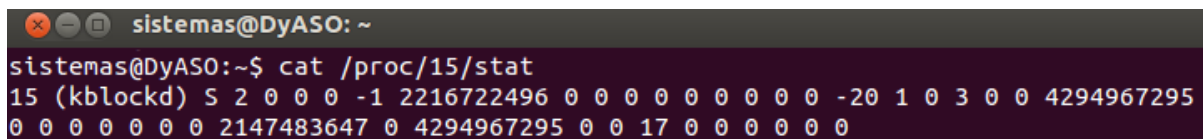
2.2. Cálculo de los porcentajes de uso de la CPU

Esta parte se va a encargar de calcular el porcentaje del uso de la CPU global y de los procesos. Dentro del código se ubica en distintas zonas, ya que se necesitan obtener datos y calcular otros en distintos tiempos de la ejecución del programa.

Primero, se va a proceder a calcular el tiempo de uso de la CPU con los datos que se obtienen de la primera línea del fichero `/proc/stat`, esta muestra los tiempos en los distintos estados de la CPU. Para ello se mostrará mediante `cat` la información y se le pasará el resultado mediante una tubería a `awk`. Este último se encargará de coger la línea cuyo primer campo es “`cpu`”, después se sumarán del campo 2 al 9 y se guardará su resultado en la variable `tcpu1`. Se realizará lo mismo en la variable `cpuUsage1`, pero excluyendo en la suma el campo número 5, el tiempo `idle`, que simboliza el tiempo de espera de la CPU y sin operaciones de E/S.

```
tcpu1=$(cat "/proc/stat" | awk '$1 == "cpu" {print ($2+$3+$4+$5+$6+$7+$8+$9+$10)}')
cpuUsage1=$(cat "/proc/stat" | awk '$1 == "cpu" {print ($2+$3+$4+$6+$7+$8+$9+$10)}')
```

A partir de los identificadores de procesos obtenidos en el paso anterior, se va a crear un array, `tpcp1`, cuyos índices serán los propios identificadores de procesos, siempre que sigan existiendo. Cada posición del array almacenará la suma de los tiempos de uso de la CPU en modo usuario y modo núcleo. Estos tiempos se obtienen del fichero `/proc/[PID]/stat`, el cual muestra información relevante del proceso con identificador `[PID]` (en la Figura 4 se puede observar los datos que muestra dicho fichero para el proceso con identificador 15).



```
sistemas@DyASO: ~
sistemas@DyASO:~$ cat /proc/15/stat
15 (kblockd) S 2 0 0 0 -1 2216722496 0 0 0 0 0 0 0 0 -20 1 0 3 0 0 4294967295
0 0 0 0 0 0 2147483647 0 4294967295 0 0 17 0 0 0 0 0 0
```

Figura 3: Información mostrada mediante `cat /proc/15/stat`

El tiempo en modo usuario se encuentra almacenado en el campo 14, el de modo núcleo en el 15 del fichero `/proc/[PID]/stat`. Para poder obtener el valor de su suma, se actuará de manera similar a las variables `tcpu1` y `tpcpu1`. Primero se mostrará mediante `cat` el fichero mencionado y mediante una tubería se le pasará a `awk` el cual simplemente se encargará de devolver la suma de los campos citados.

Para calcular este array se recorrerá mediante un bucle la variable `pids` y para cada `pid` se irá calculando y almacenando tal como se ha indicado.

```
for pid in $pids; do
    if [ -f "/proc/$pid/stat" ]; then
        tpcpu1[$pid]=$(cat "/proc/$pid/stat" | awk '{print ($14+$15)}')
    fi
done
```

El siguiente paso es hacer una pausa de un segundo con el comando `sleep 1` y se obtienen y calcula el nombre de usuario de cada proceso, que se guarda en el array `user` y se calcula igual que los identificadores de procesos, pero en este `awk` almacena el campo número 3 obtenido del mismo fichero. También se van a guardar en un array, `state`, los estados de cada proceso. Además se hará una cuenta total, en la variable `ntask`, de estos para poder mostrarlo en la cabecera del programa (se ha añadido una variable a cada estado que va contando el tipo número de tipos de estados que hay, esto es para los estados: **ejecución**, **dormido**, **a la espera** y **zombie**).

Después se volverán a calcular los mismos datos que se realizaron antes del `sleep`. Con el mismo nombre de variables pero sustituyendo el 1 que tenían al final de sus nombres por un 2 (estos son `tcpu2`,

`cpuUsage2` y `tcpcu2`, este último transformándolo al formato necesario será el valor que tendrá el campo `TIME+`).

Ahora se calcula el tiempo de uso de la cpu en el intervalo de tiempo de las dos mediciones y se almacena en la variable `tcpcu`. Para ello a la segunda medición se le resta la primera. Este será el tiempo total de la CPU, es decir el 100 %.

```
tcpcu=$(expr $tcpcu2 - $tcpcu1)
```

Para calcular el porcentaje de uso de la CPU se utiliza una regla de tres, sabiendo cual es el 100 % habrá que calcular cuál es el porcentaje cuando se está utilizando. El tiempo que se ha estado utilizando es la resta entre `cpuUsage2` y `cpuUsage1`. Aplicando dicha regla de tres se obtiene el valor que se mostrará en la cabecera del programa. Dicho valor se almacenará en la variable `cpuUsage` y la sentencia que la calcula mediante el comando `bc` (lenguaje de precisión cálculo) es:

```
cpuUsage=$(echo "scale = 4; (($cpuUsage2-$cpuUsage1)/$tcpcu)*100" | bc)
```

A continuación, se va a calcular el porcentaje de uso e CPU de cada proceso. Mediante un bucle se recorre la variable con todos los identificadores de procesos y en el array `pcpu` se irán almacenando en cada posición, y como índices los **PIDs**, los porcentajes que se calcularán de la misma manera que `cpuUsage`. Se restarán los tiempos tomados en segundo y primer lugar de cada proceso, almacenados en las posiciones de sendos arrays, y se aplicará la misma regla de tres comparando con la variable `tcpcu`.

```
for pid in $pids; do
    if [ -f "/proc/$pid/stat" ]; then
        pcpu[$pid]=$(expr "scale=4; (($tcpcu2[$pid]}-$tcpcu1[$pid]))/$tcpcu)*100" | bc)
    fi
done
```

Para finalizar, con el siguiente código, se van a ordenar de mayor a menor los porcentajes de uso de CPU e identificadores de los procesos.

```
sortedpid=$(for k in "${!pcpu[@]}"; do
    echo $k ${pcpu["$k"]}
done | sort -rnk2 -nk1)
```

En el array `sortedpid` se van a almacenar en orden numérico de mayor a menor tanto **PIDs** como porcentajes. En dicho array se guardarán en las posiciones pares los identificadores y en los impares los porcentajes. A la hora de mostrar el resultado por la terminal de salida se explicará la manera de obtener los identificadores para poder mostrar el resultado correcto requerido.

2.3. Cálculo de la memoria

Se va a empezar explicando como obtener los datos que se mostrarán en la cabecera del programa, estos son **memoria total**, **memoria libre** y **memoria usada**. Los dos primeros se pueden encontrar en el fichero `/proc/meminfo`, en las dos primeras filas respectivamente. Se obtendrán los datos de la

misma manera que se han obtenido los anteriores: mediante una tubería se le pasará al comando `awk` el resultado de usar el comando `cat` sobre el fichero mencionado. Entonces `awk` filtrará la información y obtendrá el dato requerido. La memoria usada se obtiene restando la memoria libre a la memoria total. Estos tres datos se muestran a continuación la manera de obtenerlos:

```
memTotal=$(cat /proc/meminfo | awk '/MemTotal/ {print $2}')
memFree=$(cat /proc/meminfo | awk '/MemFree/ {print $2}')
memUsed=$(expr "$memTotal - $memFree" | bc)
```

Para conocer el porcentaje de uso de memoria de cada proceso se van a necesitar obtener el **tamaño de página** y el **número de páginas por proceso**.

Para calcular el **tamaño de página** simplemente es dividir el total de memoria mapeada, que se encuentra en `/proc/meminfo` en la línea cuyo primer campo es `mapped`, entre el número de página dedicado a tal fin, cuyo dato se puede localizar en `/proc/vmstat` en la línea que empieza por `nr_mapped`. Ambos datos se obtienen de la misma manera que e han explicado los anteriores.

```
mapped=$(cat /proc/meminfo | awk '/Mapped/ {print $2}')
nrMapped=$(cat /proc/vmstat | awk '/nr_mapped/ {print $2}')
pageSize=$(expr "$mapped / $nrMapped" | bc)
```

En el siguiente paso, se ha de calcular el número de páginas utilizadas por cada proceso, obteniéndose dicho dato de el campo número 24, con nombre `rss`, de `/proc/[PID]/stat`. Con todos estos datos obtenidos se puede proceder a calcular el porcentaje de uso de memoria de cada proceso. Para cada proceso se va a realizar una regla de tres: sabiendo que la **memoria total** es el 100 %, se calculará el porcentaje para la cantidad de memoria que usa cada proceso. Esta cantidad se calcula multiplicando el **número de páginas del proceso** por el **tamaño de página**. Se guardarán estos porcentajes en el array `pmen` en cuyos índices serán los identificadores de los procesos.

```
for pid in $pids; do
    if [ -f "/proc/$pid/stat" ]; then
        rss=$(cat "/proc/$pid/stat" | awk '{print $24}') # Numero de paginas del proceso
        pmen[$pid]=$(expr "scale=4; (($rss*$pageSize)/$memTotal)*100" | bc)
    fi
done
```

2.4. Obtención del resto de datos

El resto de datos necesarios se van a obtener de `/proc/[PID]/stat`. En este caso, y para realizar un programa más eficiente en tiempo de ejecución, no se van a extraer los datos con el comando `awk`. Directamente, para cada identificador de proceso, se van a guardar en un array con la siguiente sentencia que utiliza el comando `cat`, la cual irá guardando cada campo que lea del fichero en una posición del array:

```
statPid=($pid $(cat "/proc/$pid/stat"))
```

En dicho array, para que coincidan el número de campo del fichero con el índice del array, se introduce como primer elemento el identificador de proceso al principio (esto es la posición 0).

Este array, que se calcula para cada proceso, se va a crear en el mismo bucle que realizará la impresión por pantalla de los datos.

2.5. Impresión del resultado

Cada línea de datos que se va a imprimir lleva un formato determinado que se le ha dado mediante el comando `printf`. Al cual anteriormente se le asigna la variable de ambiente con el idioma local con la sentencia `LANG=C`, esto se hace para que no de errores por ejemplo al leer una variable de coma flotante cuyos dígitos decimales estén separados de la parte entera por comas.

Lo primero que se va a imprimir va a ser la cabecera del programa. Se compondrá de tres líneas: la primera mostrará el número de procesos seguido de el número que están en cada uno de los estados: ejecutándose, dormido, espera y zombie; la siguiente línea mostrará el porcentaje de uso de la CPU que se ha utilizado mientras se ha ejecutado el script; y por último se expone la cantidad de memoria total, seguida de la memoria usada y la libre.

```
LANG=C printf "Task: %5d, %3d running, %3d sleeping, %3d stopped, %3d zombie\nCPU(s):
               %-.1f%%\nMem:      %dk total,   %dk used,   %dk free\n" $ntask $running
               $sleeping $stopped $zombie $cpuUsage $memTotal $memUsed $memFree
```

En la siguiente línea se imprime una cabecera en la que se muestra el nombre de cada campo de los procesos que se muestran que tendrá cada columna. Para que se parezca más al comando `top` se le ha cambiado el color de fondo, a blanco, y el de las letras, a negro. Esto se realiza indicando al principio de la cadena de caracteres que se desea imprimir con `\e[47;30m` (47 simboliza que se cambie el fondo a blanco y 30 las letras a negro), al final de la cadena a imprimir se introduce `\e[0m` para que vuelva a imprimir en el mismo color que siempre. Además se imprimirá hasta final de línea sea cual sea el tamaño de la terminal, esto se realiza añadiendo a la cadena a imprimir `\x1B[K`. Se imprimirá con el comando `echo -e`, donde la opción `-e` se utiliza para que se puedan imprimir las letras en colores, si no se pusiera imprimiría tal cual la cadena de caracteres.

```
echo -e '\e[47;30m  PID USER          PR   VIRT  S   %CPU  %MEM    TIME+  COMMAND\x1B[K\e[0m'
```

Se va a crear una variable, `aux`, con el valor de la longitud del array `sortedpid` menos dos, cuando se utilice se explicará el porqué.

Por último se va a recorrer un bucle de 0 a 18, inclusive, en el cual la variable que llevará la cuenta empezará con el valor cero y se irá autoincrementando en dos unidades cada vez. Así se podrá acceder a los identificadores de procesos ordenados según el porcentaje de uso de la CPU en el array creado anteriormente `sortedpid`. Accediendo a los diferentes índices se irá extrayendo los identificadores de procesos. Seguidamente, ya que los identificadores están ordenados, en primer orden por su porcentaje de uso de CPU y seguido por el PID de mayor a menor número, para parecerse más en lo posible a `top` (este muestra de arriba a abajo cuando los procesos tienen el mismo porcentaje de uso de CPU por orden creciente de sus identificadores) comprueba si el porcentaje de CPU de dicho proceso es igual a 0, si es así se coge el proceso guardado en la posición `aux` y se decrementa en dos unidades el valor de dicha (para eso era la variable mencionada en el párrafo anterior).

Ahora a partir de cada identificador guardado en la variable PID se crea el array `statPid` mencionado más arriba.

Para finalizar se dará formato a una cadena que irá mostrando por pantalla todos los datos, que irán impresos cada uno debajo de la columna con el nombre que indica su cabecera. Primero se imprime el **PID**, que se obtiene de la variable con el mismo nombre; seguido de **USER**, que se saca del array `user`; después la prioridad del proceso **PR**, se imprime de la posición 18 del array `statPid` (este valor se le pasa como parámetro a la función `setPriority()` creada para que en el caso de que su valor sea -100 imprima las letras **RT**, *real time*, tal como lo hace `top`); el cuarto campo, **VIRT**, se completa con el elemento 23 del array `statPid` dividido entre 1024 (esto se hace para pasar el valor de bytes obtenido a megabytes) además si el valor es mayor a 99999 se dividirá el valor entre 1000 y se le añadirá una **m** al final para simbolizar que el valor es por mil, esto se hace con una función creada llama `setInMiles()`; el quinto y sexto campo se mostrarán los porcentajes de uso de cpu, **%CPU**, y de la memoria, **%MEM**, almacenados en los vectores `pcpu` y `pmem` respectivamente; el siguiente campo será **TIME+**, cuyo valor lo dan la suma de los tiempos de uso de CPU en modo usuario y núcleo tomados la última vez que se encuentran en `tpcpu2`, como el valor es en nanosegundos hay que pasarlo a formato **mm:ss.cc**, esta operación la realiza la función `convertSec()`; y por último se imprime el campo **COMMAND** el cual viene de la posición con índice 2 del vector `statPid` que contiene el nombre del proceso ejecutado, este está contenido entre paréntesis, para quitarlos se usa la función `removeBrackets()` que utiliza el comando `sed` para eliminar dichos caracteres.

```
for ((i=0; i<19; i=$((i+2)));do
    pid=${sortedpid[$i]}
    if [ ${pcpu[$pid]} == 0 ]; then
        pid=${sortedpid[$aux]}
        let aux=aux-2
    fi
    if [ -f "/proc/$pid/stat" ]; then
        statPid=($pid $(cat "/proc/$pid/stat"))
        LANG=C printf "%5s %-10s%4s %6s %s %5.1f %5.1f %9s %s\n" $pid ${user[$pid]}
            'setPriority ${statPid[18]}' 'setInMiles
            $(expr ${statPid[23]} / 1024)' ${state[$pid]}
            ${pcpu[$pid]} ${pmem[$pid]} 'convertSec ${tpcpu2[$pid]}'
            'removeBrackets ${statPid[2]}'
    fi
done
```

Todas las funciones que se mencionan en el párrafo anterior además de todo el resto comentado se pueden ver en el código fuente de las líneas 6 a 36 en la Sección 4.

En la siguiente sección en la Figura 4 se puede ver un ejemplo del script ejecutado y todos los datos que se muestran explicados en esta.

3. Ejecución de ejemplo

La ejecución del programa mostrará por pantalla como resultado una imagen similar a la de la Figura 4, dependiendo de los procesos en el momento de iniciar el script.

En esta sección se van a mostrar dos ejecuciones del programa y comparar sus resultados con `top`. En la primera se comparará llamando al programa sin ningún proceso llamado en segundo plano por el alumno, la segunda será ejecutando el script `Ejercicio1.sh`, que ejecuta el comando `yes` en segundo plano. Cabe señalar que algunos datos como el porcentaje de uso de la CPU, el tiempo de ejecución o el estado no son exactamente iguales a los de `top` ya que depende de el momento exacto que se ejecutan concurrentemente y el tiempo en el que se extraen los datos explicados en el apartado anterior.

```
sistemas@DyASO: ~$ ~/DyASO_PED1_Pastor_Sanz_David/Trabajo1/mitop.sh
Task: 140, 0 running, 140 sleeping, 0 stopped, 0 zombie
CPU(s): 48.1%
Mem: 507440k total, 475312k used, 32128k free
```

PID	USER	PR	VIRT	S	%CPU	%MEM	TIME+	COMMAND
25662	sistemas	20	6504	S	2.6	0.3	0:00.07	mitop.sh
1	root	20	3528	S	0.0	0.4	0:00.41	init
2	root	20	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	S	0.0	0.0	0:01.10	ksoftirqd/0
5	root	20	0	S	0.0	0.0	0:00.40	kworker/u:0
6	root	RT	0	S	0.0	0.0	0:00.00	migration/0
7	root	RT	0	S	0.0	0.0	0:00.08	watchdog/0
8	root	0	0	S	0.0	0.0	0:00.00	cpuset
9	root	0	0	S	0.0	0.0	0:00.00	khelper
10	root	20	0	S	0.0	0.0	0:00.00	kdevtmpfs

Figura 4: Ejecución del script `mitop.sh`

3.1. Comparando mitop con top

```
top - 19:15:00 up 22 min, 3 users, load average: 0.00, 0.05, 0.08
tasks: 143 total, 3 running, 140 sleeping, 0 stopped, 0 zombie
Cpu(s): 24.8%us, 45.0%sy, 0.0%ni, 30.2%id, 0.0%wa, 0.0%ht, 0.0%st, 0.0%xs
Mem: 507440k total, 482960k used, 24480k free, 60196k buffers
Swap: 0k total, 0k used, 0k free, 189740k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
24867	sistemas	20	0	6472	1412	1184	S	4.0	0.3	0:00.12	mitop.sh
1340	root	20	0	96984	52m	11m	S	1.3	10.5	0:11.20	Xorg
2102	sistemas	20	0	66768	15m	10m	S	1.3	3.2	0:04.45	gnome-terminal
3	root	20	0	0	0	0	R	0.7	0.0	0:00.68	ksoftirqd/0
1780	sistemas	20	0	112m	24m	18m	S	0.7	5.0	0:01.03	unity-2d-panel
1797	sistemas	20	0	50676	9384	7432	S	0.7	1.8	0:02.18	banfdaemon
1732	sistemas	20	0	4540	2072	628	S	0.3	0.4	0:01.68	dbus-daemon
1762	sistemas	20	0	121m	13m	10m	S	0.3	2.6	0:01.71	metacity

```
sistemas@DyASO: ~$ ~/DyASO_PED1_Pastor_Sanz_David/Trabajo1/mitop.sh
Task: 140, 0 running, 140 sleeping, 0 stopped, 0 zombie
CPU(s): 61.3%
Mem: 507440k total, 482984k used, 24456k free
```

PID	USER	PR	VIRT	S	%CPU	%MEM	TIME+	COMMAND
24867	sistemas	20	6508	S	4.1	0.3	0:00.12	mitop.sh
1340	root	20	96984	S	1.4	10.5	0:11.20	Xorg
2102	sistemas	20	66768	S	1.0	3.2	0:04.45	gnome-terminal
1797	sistemas	20	50676	S	0.7	1.8	0:02.18	banfdaemon
1780	sistemas	20	115m	S	0.7	5.0	0:01.03	unity-2d-panel
1789	sistemas	20	119m	S	0.3	4.7	0:01.06	nautilus
1762	sistemas	20	124m	S	0.3	2.6	0:01.71	metacity
1732	sistemas	20	4540	S	0.3	0.4	0:01.68	dbus-daemon
3	root	20	0	S	0.3	0.0	0:00.68	ksoftirqd/0
1	root	20	3532	S	0.0	0.4	0:00.44	init

Figura 5: Comparación de `top` con `mitop`

En la Figura 5 se puede observar la ejecución concurrente del comando del sistema y el script realizado.

Se va a empezar cotejando la cabecera del programa. Se han subrayado con distintos colores cada dato que se va a comparar de ambos.

- Subrayado de color verde se puede observar el **número de procesos** que hay en el momento y el número de cada uno de sus estados. Esta información se puede observar que difiere de tres unidades en el total y en el tipo de estados, esto es, como se ha comentado anteriormente, porque no se toman las medidas de **top** y **mitop** en el mismo instante y hay ciertos procesos que duran pocos ciclos de reloj o que empiezan o acaban momentos antes o después de la llamada.
- En color amarillo se muestra el **porcentaje de uso de la CPU**. Más o menos concuerda con la suma de los campos de porcentaje de uso en modo usuario y núcleo de **top**
- Por último y en color azul se ve la **memoria total**, el total en ambos es obvio que será igual; y la **libre** y **usada** que difieren en pocas unidades.

Ahora se van a comparar los distintos campos que se muestran de cada proceso, en la Figura 5 los mismos campos de cada programa están recuadrados del mismo color. Se va a comprobar los datos que están en la primera línea, se puede observar que los demás son prácticamente idénticos, en otros difieren en la posición porque los porcentajes son distintos y algunos son procesos completamente distintos.

- Recuadrado en color rojo se encuentran el **PID**, **usuario** y nivel de **prioridad**. En ambos programas se encuentran los mismos datos.
- Encerrados en rectángulos de color verde están la cantidad de **memoria virtualizada**, que puede ser un poco distintas al igual que se ha explicado anteriormente, tal como se ve en la imagen; y el **estado** que en este caso son iguales, pero también podrían aparecer distintos por el momento de la captura del dato (el proceso con identificador número 3 por ejemplo en la Figura 5 muestra en **mitop** el estado de durmiendo y en **top** el de corriendo).
- En recuadros violeta están los **porcentajes de uso de CPU** y de **memoria**. En el campo asignado al de CPU se puede ver una mínima diferencia de los porcentajes, los de memoria son totalmente idénticos.
- Por último, y en cuadros amarillos, se ven el tiempo desde que el proceso empezó su ejecución, en el campo **TIME+**, que se explicó en la anterior sección y que marcan casi el mismo tiempo ambos (**top** y **mitop**); y el nombre del comando que ejecuta el proceso, que se denota por **COMMAND**, que en ambos casos son iguales.

3.2. Comparando Ejercicio1 con top

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
28815	sistemas	20	0	5424	280	224	R	60.4	0.1	0:03.55	yes
28816	sistemas	20	0	6472	1412	1184	S	1.7	0.3	0:00.07	mitop.sh
2106	sistemas	20	0	66768	15m	10m	S	0.3	3.2	0:01.58	gnome-terminal
1	root	20	0	3664	2016	1276	S	0.0	0.4	0:00.41	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	R	0.0	0.0	0:00.59	ksoftirqd/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.04	kworker/0:0
5	root	20	0	0	0	0	S	0.0	0.0	0:00.38	kworker/u:0
6	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/0

Figura 6: Comparación de **top** con **Ejercicio1**

Como en el ejemplo anterior, se van a comparar la ejecución de **top** con el script **Ejercicio1.sh**, que lanza en segundo plano **yes** t luego ejecuta **mitop.sh**. El resultado obtenido en este ejemplo se ve en la Figura 6. Como en la sección 4.1 se denotaran con colores los datos a comparar.

- En color verde se encuentran el **número de procesos** y sus estados que se encuentran. Es un poco distinta, al igual que en el ejemplo anterior. Cabe señalar que ahora se encuentran dos procesos en estado ejecutándose, uno de ellos es **yes**.
- El **porcentaje de uso de la CPU**, en amarillo. Se observa que difiere en un infimo porcentaje con la suma usuario y sistema de **top**.
- En azul está la **memoria total, libre y usada**. Se toma la misma explicación que en el ejemplo de **mitop**.

Se pasa ahora a argumentar los datos de los procesos, aunque serán igual que los dichos más arriba con **mitop**.

- El **PID**, **usuario** y nivel de **prioridad** están recuadrados en rojo. En ambos programas se encuentran los mismos datos.
- Los campos de **memoria virtualizada** y **estado**, en color verde.
- Violeta son los **porcentajes de uso de CPU** y de **memoria**. Puede haber una pequeña diferencia en estos datos.
- En cuadros amarillos están el tiempo que lleva el proceso en ejecución, en el campo **TIME+**, y el nombre del comando que ejecuta el proceso, **COMMAND**.

4. Código fuente

```

1 #!/ bin / bash
2
3 # -----
4 # Funciones y variables
5 # -----
6 # Funcion que se crea para indicar que un numero mayor a 999999 es igual a ese numero
   dividido entre 1000 y con una m al final
7 function setInMiles() {
8   if [ $1 -gt 99999 ]; then
9     var=$(expr $1 / 1000)
10    echo $var"m"
11   else
12     echo $1
13   fi
14 }
15
16 # Funcion para imprimir RT (real time) cuando la prioridad es -100
17 function setPriority() {
18   if [ $1 == -100 ]; then
19     echo "RT"
20   else
21     echo $1
22   fi
23 }
24
25 # Funcion para pasar segundos a formato mm:ss.cc. Como se le pasara como parametro tics
   de reloj , se pasara a segundos
26 function convertSec() {
27   ss=$(expr "scale=2;$1/'getconf CLK_TCK' " | bc)
28   mm=$(expr "$ss / 60" | bc)
29   ss=$(expr "scale=2;($ss-($mm*60.0))" | bc)
30   LANG=C printf "%d:%05.02f" $mm $ss
31 }
32
33 # Funcion para eliminar los parentesis del campo command. Se utiliza el comando sed
34 function removeBrackets() {
35   echo $(echo $(echo $1 | sed 's/(//g') | sed 's/)//g')
36 }
37
38 # Variable que guardara el numero de procesos. Ademas se crea una variable para cada uno
   de los posibles estados y llevar su cuenta
39 ntask=0
40 sleeping=0
41 running=0
42 stopped=0
43 zombie=0
44
45 # Variable que guarda el uso total de la cpu
46 cpuUsage=0
47
48 # -----
49 # Obtencion de datos
50 # -----
51
52 # Se obtienen los pids y usuarios de todos los procesos y se guardan en una variable
53 pids=$(ls -l /proc | awk '$9 ~ /[0-9]/ {print $9}')
54
55 # Se mide el tiempo actual de uso de la cpu. Se obtiene mediante la suma de todos los
   estados de la cpu (tuser+tnice+tsystem+tidle+tiowait+tirq+tsoftirq+tsteal+tguest)
56 tcpu1=$(cat "/proc/stat" | awk '$1 == "cpu" {print ($2+$3+$4+$5+$6+$7+$8+$9+$10)}')
57
58 # Se obtienen de /proc/[PID]/stat los tiempos de uso de la cpu en modo usuario y nucleo

```

```

    y se suman
59 for pid in $pids; do
60     if [ -f "/proc/$pid/stat" ]; then
61         tpcpu1[$pid]=$ (cat "/proc/$pid/stat" | awk '{print ($14+$15)}')
62     fi
63 done
64
65 # -----
66 # Calculo porcentaje cpu
67 # -----
68 # Para calcular el uso total de cpu se realiza la misma suma que para tpcu1 pero sin el
    tiempo de idle
69 cpuUsage1=$(cat "/proc/stat" | awk '$1 == "cpu" {print ($2+$3+$4+$6+$7+$8+$9+$10)}')
70
71 # Se duerme durante 1 segundo
72 sleep 1
73
74 # Se obtienen el usuario de cada proceso, numero de procesos, sus estados y numero de
    tipos
75 for pid in $pids; do
76     if [ -f "/proc/$pid/stat" ]; then
77         user[$pid]=$ (ls -l /proc | awk '$9 == '$pid' {print $3}')
78         let ntask=ntask+1
79         state[$pid]=$ (cat "/proc/$pid/stat" | awk '{print $3}')
80         if [ ${state[$pid]} = 'R' ]; then
81             let running=running+1
82         elif [ ${state[$pid]} = 'S' ]; then
83             let sleeping=sleeping+1
84         elif [ ${state[$pid]} = 'T' ]; then
85             let stopped=stopped+1
86         elif [ ${state[$pid]} = 'Z' ]; then
87             let zombie=zombie+1
88         fi
89     fi
90 done
91
92 # Se obtienen los tiempos de uso de la cpu en modo usuario y nucleo y se suman de nuevo
93 for pid in $pids; do
94     if [ -f "/proc/$pid/stat" ]; then
95         tpcpu2[$pid]=$ (cat "/proc/$pid/stat" | awk '{print ($14+$15)}') # Valor que tendra el
            campo TIME+
96     fi
97 done
98
99 # Se mide el tiempo actual de nuevo. Se le resta el tiempo medido anteriormente para
    poder asi hallar el tiempo de uso de la cpu.
100 tcpu2=$(cat "/proc/stat" | awk '$1 == "cpu" {print ($2+$3+$4+$5+$6+$7+$8+$9+$10)}')
101 tcpu=$((expr $tcpu2 - $tcpu1))
102
103 # Se obtiene la segunda muestra y se calcula el porcentaje de uso de la cpu
104 cpuUsage2=$(cat "/proc/stat" | awk '$1 == "cpu" {print ($2+$3+$4+$6+$7+$8+$9+$10)}')
105 cpuUsage=$((echo "scale = 4; (($cpuUsage2-$cpuUsage1)/$tcpu)*100" | bc))
106
107 # Se obtiene el porcentaje de cpu
108 for pid in $pids; do
109     if [ -f "/proc/$pid/stat" ]; then
110         pcpu[$pid]=$ (expr "scale=4; ((${tpcpu2[$pid]}-${tpcpu1[$pid]})/$tcpu)*100" | bc)
111     fi
112 done
113
114 # Se ordena el array donde se encuentran los porcentajes de cpu mediante su pid
115 sortedpid=$(for k in "${!pcpu[@]"; do
116     echo $k ${pcpu["$k"]}
117     done | sort -rnk2 -nk1))

```

```

118
119 # -----
120 # Calculo de memoria
121 # -----
122 # Se obtienen la memoria total, libre y usada
123 memTotal=$(cat /proc/meminfo | awk '/MemTotal/ {print $2}')
124 memFree=$(cat /proc/meminfo | awk '/MemFree/ {print $2}')
125 memUsed=$(expr "$memTotal - $memFree" | bc )
126
127 # Para el calculo del porcentaje de memoria por proceso primero se calcula el tamaño de
    pagina
128 mapped=$(cat /proc/meminfo | awk '/Mapped/ {print $2}')
129 nrMapped=$(cat /proc/vmstat | awk '/nr_mapped/ {print $2}')
130 pageSize=$((mapped/$nrMapped))
131
132 # Para calcular el porcentaje de memoria usada se divide el número de páginas que usa el
    proceso en memoria por el tamaño de la página dividido entre la memoria total.
    Además se calculan el número total de procesos junto con el array que indica sus
    estados y el número de estados de estos
133 for pid in $pids; do
134     if [ -f "/proc/$pid/stat" ]; then
135         rss=$(cat "/proc/$pid/stat" | awk '{print $24}') # Número de páginas del proceso
136         pmem[$pid]=$((expr "scale=4;(($rss*$pageSize)/$memTotal)*100" | bc))
137     fi
138 done
139
140 # -----
141 # Impresión
142 # -----
143 # Número de procesos, uso total de la cpu, memoria total, utilizada y libre.
144 LANG=C printf "Task: %5d, %3d running, %3d sleeping, %3d stopped, %3d zombie\nCPU(s):
    %-.1f%%Mem:      %5k total,      %5k used,      %5k free\n" $ntask $running $sleeping
    $stopped $zombie $cpuUsage $memTotal $memUsed $memFree
145
146 # Cabecera con fondo blanco y letras negras
147 echo -e '\e[47;30m PID USER          PR   VIRT  S  %CPU  %MEM      TIME+  COMMAND\x1B[K\e
    [0m'
148
149 # Se recorre desde 0 a 19 de dos en dos y se saca el pid del array ordenado
    anteriormente. En las posiciones pares se encuentran los pids y en las impares los
    porcentajes de uso de cpu de estos
150 aux=${#sortedpid[@]}-2
151 for ((i=0; i<20; i=$((i+2)));do
152     pid=${sortedpid[$i]}
153     # Para parecerse mas a top, si el porcentaje de cpu es 0, se imprimiran de menor a
    mayor por pid. En el array sortedpid estan ordenados de mayor a menor
154     if [ ${pcpu[$pid]} == 0 ]; then
155         pid=${sortedpid[$aux]}
156         let aux=aux-2
157     fi
158     if [ -f "/proc/$pid/stat" ]; then
159         # Se obtienen los datos restantes de proc/[pid]/stat. Se introducen en un array en el
            cual se introduce en la posición 0 el pid otra vez para que así al obtener los datos
            después de dicho array estén en las mismas posiciones que en stat al acceder con
            awk
160         statPid=($pid $(cat "/proc/$pid/stat"))
161         LANG=C printf "%5s %10s%4s %6s  %s %5.1f %5.1f  %9s  %s\n" $pid ${user[$pid]} '
            setPriority  ${statPid[18]} ' 'setInMiles $(expr ${statPid[23]} / 1024)' ${state[$pid]}
            ${pcpu[$pid]} ${pmem[$pid]} 'convertSec ${tpcpu2[$pid]} ' 'removeBrackets ${
            statPid[2]} '
162     fi
163 done

```


5. Opinión personal y material utilizado

La dificultad de la práctica sin haber tocado nunca antes **Bash** es, bajo el punto de vista personal del anuncio, complicada respecto a la sintaxis que difiere de otros lenguajes de programación conocidos por este. Se ha tenido algún problema con los espacios, por ejemplo, por la costumbre que se tenía al dejarlos siempre en las asignaciones. También ha costado entender algunos comandos como el método de ordenación, **sort**, que hay que hacer algo fuera de lo convencional para ordenar un vector. De todas maneras después de “pelearse” y practicar se ha hecho con el control gracias a documentos encontrados en la red, los cuales se pueden ver en la sección 6, y ha sido bastante llevadera.

Respecto a la obtención de los datos y los cálculos se ha tenido que leer el **man** de **/proc** y buscar en internet varias veces para al final comprender el funcionamiento, el cómo y el porqué de dichos valores.

En un principio la práctica se hace un poco “aburrida” porque es simular algo que el SO ya contiene, **top**, pero al final se hace bastante didáctica y ayuda a comprender algunos conceptos de la asignatura e incluso de otras.

El material utilizado ha sido la máquina virtual ofrecida por el equipo docente. Para la edición del script se ha utilizado **gedit**. El texto base y los enlaces a manuales que se incluyen en el enunciado de la práctica, más otros que el alumno ha tenido que buscar, han sido de ayuda para aprender a manejarse con **Bash scripting**. Este documento ha sido creado con el editor de texto **MiTeX** escrito en **L^AT_EX**.

6. Bibliografía

<http://man7.org/linux/man-pages/man5/proc.5.html>

<http://www.gnu.org/software/bash/manual/bash.pdf>

<http://stackoverflow.com/>

<https://www.gnu.org/software/>

<http://www.linux-es.org/node/31>