Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

fSO

Práctica 3
Monitorización de procesos en LINUX

Contenido

1. Objetivos

El objetivo de esta práctica es capacitar al alumno para observar el comportamiento de los procesos que hay en el sistema (p.ej. PID, consumo de CPU, ocupación de memoria, etc.).
Para ello, a lo largo de la sesión, los estudiantes practican el uso de las siguientes herramientas:

- Comandos de gestión de procesos (ps y top) y señales (kill).
- Contenido del directorio /proc donde Linux ofrece toda la información sobre la ejecución del proceso.
- Herramientas para automatizar informes de monitorización y control de procesos:
    - Filtrado de texto (grep).
    - Programación de Shell.

2. Utilidades y órdenes del shell para procesos

Cree un nuevo directorio denominado, por ejemplo, fso_pract3, con la orden **mkdir**, para realizar esta práctica:

```
$ mkdir fso_pract3
$ cd fso_pract3
```

2.1. Orden ps

La orden **ps** permite listar información sobre la ejecución de los procesos en Linux.

```
$ ps u
  USER      PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
  gandreu   251   0,0  0,0  2435492   1112 s000  S     4:20PM   0:00.01 -bash
```

El comando **ps** muestra la lista de procesos activos basándose para ello, en la información del directorio de sistema **/proc**. Dependiendo de los argumentos utilizados con **ps** se muestra más o menos información, distribuida en un formato de columnas. A continuación, se muestra una lista de las columnas más habituales:

PID	Número de identificador del proceso.
PPID	Número de identificador del padre del proceso
UID	Usuario propietario del proceso
TTY	Terminal asociada al proceso. Si no hay aparece un ?
TIME	Tiempo de uso de CPU acumulado por el proceso
CMD	El nombre del programa o comando que inició el proceso
RSS	Resident Size, tamaño de la parte residente en memoria en kilobytes
SIZE	Tamaño virtual de la imagen del proceso
NI	Valor Nice (prioridad) del proceso. Cuanto menos valor, más prioridad.
PCPU	Porcentaje de CPU utilizado por el proceso
STIME	Starting time. Hora de inicio del proceso
STAT	Estado del proceso.

En la columna STAT se muestra el estado del proceso, que puede ser:

- R (Runnable): En ejecución.
- S (Sleeping): Proceso en espera de algún evento para poder continuar.
- T (sTopped): Proceso detenido totalmente. Puede ser reiniciado posteriormente.
- Z (Zombie): Proceso zombie.
- D (uninterruptible sleep): Proceso en espera ininterrumpible, asociados a acciones de entrada/salida.

A su vez, junto con la letra que indica el estado del proceso, pueden aparecer caracteres adicionales que dan más información sobre el estado de un proceso. Los más habituales son:

- +: El proceso está ejecutándose en primer plano.
- X : Proceso que está siendo depurado.

Por defecto, el comando **ps** muestra la información referente a PID, TTY, TIME y CMD. Ejecutándolo con diferentes opciones podemos controlar cuanta información se mostrará.

1.Ejercicio: *Ejecute los siguientes comandos y observe las diferencias en sus salidas.*

$ps
$ps u
$ps -la
$ps f
$ps aux

Habitualmente **ps** se utiliza conjuntamente con la orden **grep** que se explica más adelante.

2.2. Orden top

La orden **top** muestra en tiempo real un listado de los procesos activos en el sistema, especificando el % de CPU y memoria utilizado, sus PID's, usuarios, etc. Ejecute la orden `top` y en pantalla le aparecerá la información dividida en dos partes:

```
Processes: 72 total, 2 running, 1 stuck, 69 sleeping, 322 threads                                                 16:58:05
Load Avg: 0.15, 0.17, 0.16  CPU usage: 2.87% user, 3.34% sys, 93.77% idle  SharedLibs: 16M resident, 11M data, 0B linkedit.
MemRegions: 11390 total, 385M resident, 39M private, 431M shared.
PhysMem: 627M wired, 936M active, 193M inactive, 1756M used, 2340M free.
VM: 164G vsize, 1123M framework vsize, 84839(0) pageins, 0(0) pageouts. Networks: packets: 7535/6626K in, 5609/1252K out.
Disks: 22675/1057M read, 12667/355M written.

PID  COMMAND      %CPU TIME     #TH  #WQ  #POR #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE  PGRP PPID STATE    UID  FAULTS  COW
288  Grab         3.8  00:01.72 5    4    108- 211+  4088K+ 40M-   15M+   29M+   2483M+ 288  143  sleeping 504  13620+  366
286  quicklookd   0.0  00:00.05 4    1    72   73    2468K  7384K  6016K  534M   2920M  286  143  sleeping 504  1921    228
281- CVMCompiler  0.0  00:00.03 1    0    27   38    904K   460K   1892K  18M    591M   281  143  sleeping 504  622     81
280  xpchelper    0.0  00:00.02 2    2    39   41    972K   220K   4468K  30M    2390M  280  1    sleeping 504  1401    168
278  mdworker     0.0  00:00.12 3    1    55   63    1392K  9564K  5584K  23M    2411M  278  142  sleeping 89   2424    202
```

En la parte superior del terminal se visualiza la información global del sistema, como el número de procesos y tipo; la carga y utilización de la CPU, utilización de la memoria y discos. A continuación, aparece un listado de procesos, que pueden ser ordenados por uso de CPU o memoria, lo que es una **excelente ayuda para detectar procesos que consumen excesivos recursos**. Este listado, muestra atributos de los procesos, como pueden ser el PID de proceso, usuario que lo ejecuta, porcentaje de CPU y memoria que consume, comando que está ejecutando y tiempo de ejecución del proceso entre otros.

¡Atención! Para salir de la orden **top,** hay que teclear q (quit).

2.3. Orden kill

El comando **kill** envía señales (*signals*) a los procesos. La acción por defecto que ocasiona el envío de una señal a un proceso, es finalizar o matar el proceso. La sintaxis de esta orden es:

```
$ kill SIGNAL  PID
```

Donde SIGNAL representa la señal a enviar y PID el número de ID del proceso al cual se le entrega la señal. Obtenga el listado de todas las señales de su máquina, para ello ejecute:

```
$ kill -l
```

```
ivanovic:~ gandreu$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGURG
17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGINFO     30) SIGUSR1     31) SIGUSR2
```

De estas señales, las más interesantes son:

- SIGINT (2) que permite interrumpir (terminar de forma abrupta) un proceso y que se puede enviar al proceso que está en primer plano con la combinación de teclas CTRL-C;
- y la señal SIGKILL (9) que provoca la terminación del proceso, que se puede enviar con la orden **`kill` desde el `terminal`**. SIGKILL no se puede enmascarar, manejar o ignorar.

Para probar estas señales vamos a usar la orden 'yes', que muestra indefinidamente por pantalla el carácter 'y'.

2.Ejercicio: Lance la orden yes en primer plano:

```
$ yes
```

y finalice el proceso enviando la señal SIGINT tecleando la combinación CTRL-C**.** Ahora, lance la orden **`yes`** en *background*,

```
$ yes >/dev/null &
```

Compruebe, que, a diferencia de la ejecución anterior en primer plano, ahora no se muestra la 'y' en pantalla. Esto es porque se ha redireccionado la salida estándar al dispositivo nulo (/dev/null). También, al ejecutarse en segundo plano, nos devuelve el PID del proceso creado. Compruebe además que el terminal no está bloqueado a la espera de la terminación del proceso lanzado (yes). Por ejemplo, nos puede devolver algo como:

[1] 11148

Donde el número 11428 representa el PID del proceso lanzado.

A continuación, finalice el proceso **`yes`** utilizando la orden **`kill`**, después compruebe con **`ps`** que ya no existe el proceso.

```
$ kill -SIGKILL PID
```

Recuerde que en PID tiene que poner el número del proceso creado. La orden **`kill`** permite referenciar las señales por su número o nombre, por ejemplo:

```
#> kill -9 11428         (termina el proceso con PID 11428)
#> kill -SIGKILL 11428   (Lo mismo que lo anterior)
```

El comando `killall` es similar a kill SIGKILL pero indicando el nombre del programa. El comando **`killall`** finaliza todos los procesos con el nombre especificado.

3.Ejercicio: Ejecute la siguiente secuencia y compruebe el funcionamiento de **`killall`**

```
$ yes >/dev/null &
$ yes >/dev/null &
$ yes >/dev/null &
$ ps -la
```

```
….
$ killall  yes
$ ps -la
```

4. Ejercicio: *En un terminal ejecuta el editor Kate en background. Identifique el proceso creado y utilice la orden* **`kill`** *para terminar el proceso iniciado.*

3. El directorio /proc

En `/proc` el sistema mantiene toda la información de los procesos activos. Procesando la información de `/proc` podemos extraer información idéntica a la mostrada por **`ps`** o **`top`**.

NOTA: El directorio /proc es propio de algunos sistemas Unix, como Linux. Otros SO basados en Unix, como Mac OS X, no generan la información del sistema en este directorio. Ver apéndice para más información.

El directorio `/proc` contiene un subdirectorio `/proc/$pid` por cada proceso en ejecución y un conjunto de archivos con información del sistema:

- **`cpuinfo`:** Contiene información acerca del procesador.
- **`stat:`** Contiene información del rendimiento del procesador: incluye información acerca del tiempo que lleva el procesador encendido, el número de procesos en ejecución, etc.
- **`version:`** Muestra información acerca de la versión de kernel, compilador `gcc` y debian... de nuestro ordenador.

5. Ejercicio: *Ejecute la siguiente secuencia para comprobar el contenido de `/proc`. Verifique que existe un directorio cuyo nombre es el pid de su Shell. Para conocer este pid, ejecute la orden `ps` (sustituya `$pid_bash` por el PID de su shell):*

```
$ ps
…….
$ ls /proc
…
$ ls /proc/$pid_bash

$ more /proc/cpuinfo
```

En cada directorio de un proceso (`/proc/$pid`) aparecen ficheros con información de ese proceso. Algunos de ellos son:

- **`/proc/$pid/cmdline`**: Contiene nombre del proceso, incluyendo la ruta completa del mismo y los argumentos.
- **`/proc/$pid/stat:`** Se trata de un fichero de texto compuesto por varios campos, los cuales se encuentran separados por el carácter espacio. Entre los distintos campos tenemos el `Pid,` el nombre del proceso (sólo el nombre del proceso, la ruta completa no), etc.
- **`/proc/$pid/maps:`** Contiene información del mapa de memoria del proceso.
- **`/proc/$pid/status:`** Este fichero contiene información relativa al proceso como el `Pid, PPid`, estado del proceso (línea State) y atributos relacionados con el consumo de memoria.

En la siguiente figura se muestra un ejemplo de contenido de un fichero status:

- `VmSize:` Indica el tamaño total del espacio virtual que ha solicitado el proceso.
- `VmLck`: Indica el espacio ocupado por aquellas páginas que han sido bloqueadas por el proceso en su espacio virtual. El bloqueo evita que dichas páginas puedan ser seleccionadas como víctimas por el algoritmo de reemplazo.
- `VmRSS`: Proporciona el "resident set size", es decir, el tamaño del conjunto de páginas actualmente cargado en memoria. En otras palabras, nos indica cuánta memoria física está consumiendo

actualmente el proceso.

- `VmData`: Indica el tamaño virtual asignado a los datos del proceso.
- `VmStk`: Indica el tamaño virtual asignado a la pila.
- `VmExe`: Indica el tamaño virtual asignado al código del programa que ejecuta el proceso.
- `VmLib`: Indica el tamaño virtual asignado a las bibliotecas de enlace dinámico que está utilizando el proceso.

```
gandreu@shell-sisop:~$ more /proc/$$/status
Name:   bash
State:  S (sleeping)
Tgid:   10956
Pid:    10956
PPid:   10955
TracerPid:      0
Uid:    1692409199      1692409199      1692409199      1692409199
Gid:    1692452651      1692452651      1692452651      1692452651
FDSize: 256
Groups: 1692441063 1692452651 1692453521 1692453528 1692459925 1692459927 169247
8937 1692515549
VmPeak:    21168 kB
VmSize:    21168 kB
VmLck:         0 kB
VmHWM:      4108 kB
VmRSS:      4108 kB
VmData:     2672 kB
VmStk:        88 kB
VmExe:       760 kB
VmLib:      1920 kB
VmPTE:        56 kB
Threads:        1
SigQ:   0/8191
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000010000
SigIgn: 0000000000384004
SigCgt: 000000004b813efb
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
Cpus_allowed:   03
Mems_allowed:   00000000,00000001
voluntary_ctxt_switches:        88
nonvoluntary_ctxt_switches:     59
gandreu@shell-sisop:~$
```

Para obtener información exhaustiva de estos ficheros y otros, puede consultarse las páginas de manual usando el comando:

```
$ man proc
```

4. Herramientas para monitorización de procesos

4.1. Orden grep

La orden `grep` nos permite seleccionar información de un fichero. Actúa como un filtro que busca líneas que contengan una cadena. Por ejemplo:

```
$grep hola fichero      //muestra las líneas que contienen la cadena "hola" en el fichero "fichero"
$grep -i Hola fichero   //muestras las líneas con "hola" del "fichero" ignorando mayúsculas/minúsculas
$ls -l | grep pract3    //muestra las líneas con pract3 en la secuencia resultante de ejecutar ls -l.
```

6.Ejercicio: *Escriba una línea de órdenes que muestre por pantalla la frecuencia de los procesadores de su sistema. Se trata de acceder al fichero `/proc/cpuinfo` y utilizar el grep para seleccionar la línea adecuada.*

7.Ejercicio: *Escriba una línea de órdenes que muestre por pantalla el tamaño de memoria cache de los*

procesadores de su sistema. Se trata de acceder al fichero `/proc/cpuinfo` y utilizar el grep para seleccionar la línea adecuada.

4.2. Shell script

El Shell o interprete de órdenes de LINUX (p.e., `bash`) es un programa que se ejecuta a nivel de usuario, disponible en cualquier entorno LINUX y cuyo cometido es leer líneas de órdenes, analizarlas y ejecutarlas realizando las llamadas al sistema necesarias. Este intérprete define, al igual que ocurre con otros intérpretes, un lenguaje de programación propio que posee características como:

- Procedimientos, más conocidos como Shell scripts
- Palabras y caracteres reservados (también conocidos como metacaracteres)
- Variables
- Estructuras para el control del flujo tipo `if, while`, etc.
- Manejo de interrupciones

Actualmente en LINUX existen varios intérpretes de órdenes, pudiendo elegir cada usuario el que prefiera. El Shell más básico es Bourne shell o sh y está disponible en cualquier sistema. Existen otros shells, como Korn shell o ksh, el shell con sintaxis del tipo de lenguaje C o csh y el Bourne-Again SHell o bash, que es una versión extendida del sh. En esta práctica se va a utilizar el bash que es el Shell instalado en los equipos del laboratorio.

Las órdenes a ejecutar por el shell pueden ser proporcionadas desde teclado o como un programa contenido en un fichero. Se denomina **shell script** a un fichero de texto que contiene órdenes para ser ejecutadas por el shell. Por ejemplo, se puede crear un fichero que contenga:

```
#!/bin/bash
# content
Num_process=$(ls –d /proc/[1-9]*|wc –l)
echo Number of System process is: $Num_process
```

Shebang es, en la jerga de Unix, el nombre que recibe el par de caracteres #! que se encuentran al inicio del shell script anterior. En algunas ocasiones se le denomina también *hash-bang* o *sharpbang*. A continuación de estos caracteres se indica la ruta completa al intérprete de las órdenes. En nuestro caso, será "#!/bin/bash", (que es una llamada al intérprete de comandos Bash). Las líneas que comienzan por el carácter # son comentarios. El resto son sentencias que ejecutará el shell. En particular, la orden `echo` muestra lo que se le indica por pantalla. La orden `ls` sirve para obtener un listado de los ficheros directorios, mientras que "`wc –l`" cuenta el número de líneas.

8. Ejercicio: *Cree el fichero anterior usando un editor de textos y llámelo "content". A continuación, ejecútelo dándole los permisos adecuados:*

```
$ chmod +x content
$ ./content
```

Observe que con `./content` se le está indicado al shell que el fichero a ejecutar se encuentra en el directorio actual de trabajo (directorio "."). Para no tener que indicar cada vez dónde se encuentra el fichero a ejecutar (indicando la ruta completa), consulte el anexo1 de esta práctica.

Un shell script puede invocarse con argumentos, los cuales pueden ser referenciados como `$0, $1, $2, $3`, etc... El argumento `$0` referencia el propio nombre del programa, en tanto que `$@` referencia todos los argumentos y `$#` indica el número de argumentos.

9.Ejercicio: *Cree un archivo denominado `arguments` con el siguiente contenido:*

```
#!/bin/bash
# arguments
echo El número de argumentos es: $#
echo La orden completa tecleada es: $0 $@
echo La orden tecleada es: $0
echo El primer argumento es: $1
echo El segundo argumento es: $2
echo El tercer argumento es: $3
```

A continuación, ejecute `arguments` con diferentes parámetros

```
$ ./arguments uno dos tres

$ ./arguments  FSO TCO ESO
```

4.3. Variables

Una variable se crea usando "=" para asignar el identificador al contenido, "identificador=contenido". Para acceder al contenido de una variable es necesario el símbolo $ delante de su identificador. Ejemplos:

```
$ name=Alberto
$ subject=FSO
$ msg="Hello World"
…
$ echo $name
Alberto
$ echo $subject
FSO
$ echo $msg
Hello World
```

¡Cuidado! si dejamos espacios entre el = y el identificador o el valor el shell creerá que son comandos a ejecutar y no la asignación de una variable. Para acceder al contenido de una variable empleamos $ delante de su identificador:

4.4. Bucles for

La sentencia *for* itera sobre una lista de valores y asigna en cada iteración un valor a la variable asociada. Su sintaxis es la siguiente:

```
for variable in lista de valores
do
      sentencias
done
```

Podemos utilizar el bucle *for* para listar el número de argumentos del ejemplo anterior:

```
#!/bin/bash
echo El numero de argumentos es: $#
echo La orden tecleada es: $0 $@
echo Lista de argumentos:
for i in $@
do
   echo $i
done
```

La lista de valores de un bucle *for* también pueden ser los ficheros del directorio actual. Por ejemplo, el siguiente programa crea una copia de seguridad de cada uno de los ficheros del directorio actual:

```
for k in *
do
 cp $k $k.bak
 echo Creada copia de $k
done
```

Finalmente, la lista de valores de un bucle *for* también puede provenir de la ejecución de una orden mediante el uso de `$().` Por ejemplo:

```
for i in $(ls)
do
 echo $i
done
```

10.Ejercicio: *Realice un script `list_all_PID` que saque por pantalla un listado con todos los procesos del sistema. El script deberá usar un bucle for para recorrer todos los directorios que están en el directorio /proc y que empiecen por un dígito. El resultado tiene que ser como el que sigue:*

```
1
10
101
…
21
23
…
```

AYUDA: Para quedarnos solo con el nombre del fichero (que es el PID) se puede usar la orden `basename`. Para ello podemos incluir en nuestro script el siguiente código, que a partir del nombre de fichero i (con path) nos crea una variable pid con el PID del proceso:

```
pid=$(basename "$i")
```

4.5. Sentencia if

La sentencia *if* permite la ejecución condicional de órdenes. Su sintaxis es:

```
if orden se ejecuta con éxito
then
 sentencias
else
 sentencias alternativas
fi
```

Observe que la condición de la sentencia *if* no es una expresión sino una orden de LINUX. La condición es cierta si la orden "termina correctamente", (en cuyo caso se ejecutan las sentencias que siguen el *then*) y falsa

en caso contrario (en cuyo caso se ejecutan las sentencias que siguen el *else*). La cláusula *else* es opcional.

4.6. Orden test

La orden ***test*** permite evaluar condiciones y, por lo tanto, resulta de gran utilidad para utilizarla conjuntamente con la sentencia *if*. Los tipos de expresiones que puede evaluar la orden *test* son los siguientes:

Expresiones numéricas. La forma general de las expresiones es:

```
N  <primitiva>  M
```

donde N y M son interpretados como valores numéricos. Las primitivas que se pueden utilizar son:

-eq	N y M son iguales.
-ne	N y M son distintos.
-gt	N es mayor que M.
-lt	N es menor que M.
-ge	N es mayor o igual que M.
-le	N es menor o igual que M.

11.Ejercicio. Edite un archivo llamado *my_process con las siguientes órdenes y ejecútelo:*

```
#!/bin/bash
# my_process
process=$(ps u | grep $USER | wc -l)
if test $process -gt $1
then
 echo "More than $1 user processes active"
else
 echo "Equal or less than $1 user processes active"
fi
```

Expresiones alfanuméricas. Sean S y R cadenas alfanuméricas. Podemos tener dos tipos de expresiones:

```
<primitiva> S
S <primitiva> R
```

Las primitivas que se pueden utilizar son:

Primitiva	Condición
S=R	las cadenas S y R son iguales.
S != R	las cadenas S y R son distintas.
-z S	comprueba si la cadena S tiene longitud cero.
-n S	comprueba si la cadena S tiene una longitud distinta de cero

Tipos de ficheros. La forma general de las expresiones es:

```
<primitiva> fichero
```

Las primitivas que se pueden utilizar son:

Primitiva	Tipo de fichero
-s	comprueba que el fichero existe y no está vacío.
-f	comprueba que el fichero existe y es regular (no directorio).
-d	comprueba si el fichero es un directorio.
-r	comprueba si el fichero tiene permiso de lectura.
-w	comprueba si el fichero tiene permiso de escritura.
-x	comprueba si el fichero tiene permiso de ejecución.

12.Ejercicio: *Realice un script `delete_file`, que acepte el nombre de un fichero que se le pasa como argumento, al ser ejecutado, desde la línea de órdenes. Este script debe comprobar que el fichero existe y que no es de tipo directorio. Si no existe debe mostrar un mensaje de texto por pantalla; si existe y es un fichero de tipo regular debe borrarlo, mientras que si es un directorio debe mostrar un mensaje indicado que es un directorio y que no se puede borrar.*

4.7. Scripts de consulta de procesos

Para poder obtener la información más significativa de los procesos, cree un shell script *`inf_process`* con el siguiente contenido:

```
#!/bin/bash
# info proceso
pid_process=$1
ppid=$(awk '/PPid/ {print $2}' /proc/$1/status)
estado=$(awk '/State/ {print $2}' /proc/$1/status)
cmd=$(tr -d '\0' </proc/$1/cmdline)
echo $pid_process $'\t' $ppid $'\t' $estado $'\t' $cmd
```

Este script toma como argumento el PID de un proceso e imprime por pantalla en formato de columnas tabuladas el PID, PPID, ESTADO y el COMANDO que ejecuta dicho proceso. Esta información se ha obtenido de los ficheros **/proc/$pid/status**, utilizando la orden awk para obtener los valores de los campos PPid y State. La orden (cmd) se ha obtenido directamente del contenido del fichero **/proc/$pid/cmdline** utilizando la order `tr` para eliminar los 0 binarios en el texto. (Ver anexo I para consultar el funcionamiento de las ordenes awk y tr). Fíjese también en la última línea (echo): los campos están separados por un tabulador, que se codifica como `$'\t'`.

13. **Ejercicio:** *Realice un shell script denominado `inf_one_process` al que se le pasa un argumento (el PID) y saca la información de ese proceso haciendo uso del `inf_process`. El resultado será parecido al que sigue (variando el PID y comando, claro).*

```
PID    PPID  STATE       CMD
8900   8880  S           /bin/bash
```

Nótese que simplemente hay que añadir una línea de texto con la cabecera y luego llamar al script *`inf_one_process`*.

14.Ejercicio: *Realice un shell script denominado `inf_all_process` que saque un listado con la información de todos los procesos del sistema. Usar como base el script list_all_PID desarrollado anteriormente. Modificarlo para incluir la cabecera y llamar al script `inf_process`. El resultado será parecido al siguiente:*

```
PID    PPID  STATE      CMD
1      0     S          /sbin/initsplash
10     2     I
101    2     I
….

1091   1     S          /sbin/wpa_supplicant-u-s-O/run/wpa_supplicant
1092   1     S          /usr/sbin/NetworkManager--no-daemon
….
```

5. ANEXO I. Otras órdenes y filtros usadas en los scripts

5.1. Variable PATH

Para no tener que indicar cada vez dónde se encuentra el fichero a ejecutar (indicando la ruta completa), el shell utiliza la variable de entorno denominada PATH. Esta variable contiene la lista de nombres de directorios (separados por el carácter ":") donde el shell buscará los ficheros ejecutables.

Consulte el valor de la variable PATH ejecutando la orden echo:

```
$echo $PATH
```

Compruebe si el PATH que está utilizando el shell contiene el directorio actual de trabajo ("."). En caso afirmativo ejecute el fichero contenido como:

```
$contenido
```

En caso negativo, debe incluirlo ejecutando la orden:

```
$PATH=$PATH:.
```

Este cambio sólo durará hasta que finalice la ejecución del shell (o sea, se cierre la terminal). Si desea que el cambio sea permanente edite el fichero de configuración `$HOME/.bashrc` y añada al final la orden anterior.

Ejercicio Anexo1: *Edite el fichero de configuración `$HOME/.bashrc` y añada al final la orden "`PATH=$PATH:.`" Una vez añadido, abra un nuevo terminal y compruebe que el cambio se ha aplicado.*

```
$xemacs $HOME/.bashrc

$ echo $PATH
```

5.2. Operaciones aritméticas

Para que el Shell evalué una operación aritmética:

`$((expresión))`

`evalua la expresión aritmética y reemplaza el bloque por el resultado`

Por ejemplo:

```
$ echo 1+1
1+1
$ echo $((1+1))
2
```

Algunos operadores aritméticos soportados:

```
+     suma
*     mutiplicación
-     resta
/     división entera
%     resto de la división entera
( )   agrupar operaciones
```

5.3. Bucles while

Se trata de otra estructura de bucle que permite ejecutar un bloque de órdenes mientras se evalúe una condición a cierto:

```
while CONDICION; do
     bloque de comandos
done
```

Cada iteración se evalúa la condición y en el momento que no sea cierta, el bucle termina. Ejemplos de bucles:

```
# equivalente a seq 1 5 i=1
while [ $i -lt 6 ]; do
     echo $i
     i=$(($i+1))
done

# lee de stdin hasta que se introduzca 'quit' read linea

while [ "$linea" != "quit" ]; do read linea
done
```

5.4. Orden tr

Esta orden permite cambiar o traducir los caracteres procedentes de la entrada de acuerdo a reglas que se especifican. El formato general es:

```
$ tr [opciones] cadena_1 cadena_2
```

Ejemplos de utilización de este mandato son:

Para cambiar un carácter por otro: por ejemplo, el utilizado como separador entre campos de un archivo (':') con otro (por ejemplo, el tabulador):

```
$ tr : '\t' < nom_fich_entrada
```

Para cambiar un conjunto de caracteres: para poner en mayúsculas todos los caracteres que aparecen en un archivo:

```
$ tr '[a-z]' '[A-Z]' < nom_fich_entrada
```

Sustituir los caracteres nulos que contiene un fichero por blancos:

```
$ cat nom_fich_entrada | tr "\000" " " > nom_fich_salida
```

Por ejemplo:

```
tr 'Dato4' 'Dato5' < ejemplo.txt
```

5.5. Orden awk

La orden *awk* es una herramienta del sistema UNIX útil para modificar archivos, buscar y transformar conjuntos de datos, generar informes simples, etc. Puesto que es un lenguaje de programación, *awk* tiene una gran variedad de posibilidades y en esta sección sólo se ilustra su uso más básico.

La función básica de awk es buscar líneas en ficheros u otras unidades de texto que contienen ciertos patrones. Cuando en una línea se encuentra un patrón, *awk* realiza las acciones especificadas para dicho patrón sobre dicha línea. *Awk* sigue realizando el procesamiento de las líneas de entrada de esta forma hasta que llega al fin de fichero.

SINTAXIS

Awk se le ha de indicar qué patrones buscamos, qué acciones debe realizar sobre las líneas que contengan dichos patrones y como adquiere las líneas de entrada. Los patrones y acciones se le indican mediante un programa.

```
$ awk 'programa' fichero_de_entrada_1 fichero_de_entrada_2 ...
```

Este formato indica al shell que ejecute *awk* y use programa para procesar las líneas de los ficheros de entrada fichero_de_entrada_1 fichero_de_entrada_2 ...

Cuando el programa es largo es más conveniente ponerlo en un fichero y ejecutarlo de la siguiente manera:

```
$ awk –f prog-file fichero_de_entrada_1 fichero_de_entrada_2 ...
```

en este caso `prog-file` es un fichero que contiene el programa.

Un programa en *awk* consiste en una serie de reglas. Cada regla especifica un patrón a buscar, y una acción a realizar cuando se encuentre dicho patrón en el registro de entrada. La acción se encierra entre llaves para separarla de los patrones.

```
patrón { acción } patrón { acción }
...
patrón { acción }
```

PATRONES

En los patrones se pueden utilizar expresiones regulares encerradas entre barras diagonales /. La expresión regular más simple es una secuencia de letras, números o ambos. Por lo tanto, el patrón `/root/` casa con cualquier registro que contenga la cadena root.

ACCIONES

La acción da lugar a que algo suceda cuando un patrón concuerda. Si no se especifica una acción, *awk* supone
`{print}`. Esta acción copia el registro (suele ser una línea) del fichero de entrada en la salida estándar. La instrucción *print* puede ir seguida de argumentos haciendo que *awk* imprima sólo los argumentos. A continuación, se mostrarán ejemplos del uso de *awk*.

Si se genera el fichero prueba.txt del siguiente modo:

```
echo -e "Col1\tCol2\tCol3\tCol4\n" > ejemplo.txt
```

Y se visualiza con el mandato `cat`:

```
cat ejemplo.txt
```

Devolverá el siguiente contenido:

```
Col1    Col2    Col3    Col4
```

Si se utiliza *awk* para que solo muestre la columna 1 y la columna 3 del siguiente modo:

```
awk '{ print $1, $3}' ejemplo.txt
```

La salida devolverá lo siguiente:

```
Col1 Col2
```

Si se añaden datos al fichero *ejemplo.txt* del siguiente modo:

```
echo -e "Dato1\tDato2\tDato3\tDato4\n" >> ejemplo.txt
echo -e "Dato5\tDato6\tDato7\tDato8\n" >> ejemplo.txt
echo -e "Dato9\tDato10\tDato11\tDato4\n" >> ejemplo.txt
```

Y se visualiza con el mandato cat:

```
cat ejemplo.txt
```

Devolverá el siguiente contenido:

```
Col1    Col2    Col3    Col4
Dato1   Dato2   Dato3   Dato4
Dato5   Dato6   Dato7   Dato8
Dato9   Dato10  Dato11  Dato4
```

Si se utiliza nuevamente *awk* para que solo muestre la columna 3 y la columna 1(en ese orden) del siguiente modo:

```
awk '{ print $3, $1}' ejemplo.txt
```

La salida devolverá lo siguiente:

```
Col3 Col1
Dato3 Dato1
Dato7 Dato5
Dato11 Dato9
```

Si se utiliza el mandato *awk* del siguiente modo para que muestre solo la línea cuya columna contenga la expresión regular Dato5:

```
awk '/Dato5/ { print }' ejemplo.txt
```

La salida devolverá lo siguiente:

```
Dato5   Dato6   Dato7   Dato8
```

Si se utiliza el mandato *awk* del siguiente modo para que muestre solo la línea cuya columna contenga la expresión regular Dato5, y además solo las columnas 1 y 4:

```
awk '/Dato5/ { print $1, $4}' ejemplo.txt
```

La salida devolverá lo siguiente:

```
Dato5 Dato8
```

DIVIDIENDO LA ENTRADA EN REGISTROS Y CAMPOS

La entrada en el programa *awk* típico se lee en unidades llamadas registros, y éstos son procesados por las reglas uno a uno. Por defecto, cada registro es una línea del fichero de entrada.

El lenguaje *awk* divide sus registros de entrada en campos. Por defecto asume que los campos están separados por espacios en blanco. El modo en que *awk* divide los registros de entrada en campos es controlada por el separador de campos, el cual es un carácter simple o una expresión regular. *Awk* recorre el registro de entrada en búsqueda de coincidencias del separador de campos; los campos son el texto que se encuentra entre dichos separadores de campos encontrados. Por ejemplo, si el separador de campo es ':', entonces la siguiente línea:

```
maria:x:1000:1000:Maria Garcia Garcia:/home/maria:/bin/bash
```

será particionada en 7 campos:

'maria', 'x', '1000', '1000', Maria Garcia Garcia', '/home/maria' y '/bin/bash'.

El separador de campos puede ser fijado en la línea de órdenes usando el argumento ‘-F’ . Por ejemplo:

```
$ awk ‘–F:’ 'programa' ficheros_de_entrada
```

Fija como separador de campos el carácter ':'.

Para referirse a un campo en un programa *awk*, se utiliza el símbolo $ seguido por el número del campo. Por lo tanto `$1` se refiere al primer campo, `$2` al segundo y así sucesivamente. `$0` es un caso especial, representa el registro de entrada completo.

Otro ejemplo, si escribiéramos:

```
$ awk –F: '/model name/ {print $2}' /proc/cpuinfo
```

Se imprimiría el segundo campo separado por ‘:’ de las líneas que contuvieran `model name` del fichero
`/proc/cpuinfo.` Nótese que el argumento F: funciona indistintamente con y sin comillas.

PATRONES ESPECIALES BEGIN Y END

BEGIN y END son patrones especiales. Se utilizan para suministrar al awk qué hacer antes de empezar a procesar y después de haber procesado los registros de entrada. Una regla BEGIN se ejecuta una vez, antes de leer el primer registro de entrada. Y la regla END una vez después de que se hayan leído todos los registros de entrada. Por ejemplo, el siguiente mandato específica que al inicio se imprima en la salida la frase "Hola mundo" y terminar el procesamiento.

```
awk 'BEGIN { print "Hola mundo"; exit }'
```

Lo anterior deberá devolver una salida como la siguiente:

```
Hola mundo
```

Otro ejemplo:

```
$ awk 'BEGIN {print "Cuantas veces aparece Dato4" ; dato=0 ; }
>/Dato4/ { ++dato }
>END {print "Dato4 aparece " dato " veces"}' ejemplo.txt
```

Este programa averigua cuantas veces aparece la cadena ´Dato4´ en el fichero ejemplo.txt generado en

ejemplos anteriores. La regla BEGIN imprime un título para el informe e inicializa el contador dato a 0, aunque no habría necesidad ya que awk lo hace por nosotros automáticamente.
La segunda regla incrementa el valor de la variable dato cada vez que se lee de la entrada un registro que contiene el patrón´Dato4’. La regla END imprime el valor de la variable al final de la ejecución.

LA SENTENCIA IF

La sentencia `if-else` es una sentencia para la toma de decisiones de *awk*. Presenta la siguiente forma:

`if (condición) cuerpo-then [else cuerpo-else]`

donde condición es una expresión que controla qué es lo que realizará el resto de la sentencia. Si la condición es verdad, entonces se ejecuta el cuerpo-*then*; sino se ejecuta cuerpo-*else*(asumiendo que esté presente la cláusula *else*). La parte *else* de la sentencia es opcional. La condición se considera falsa si su valor es 0 o una cadena nula, sino se considerará verdadera.

Ejemplo:

```
$ awk '{ if ($0 % 2 == 0)
> print $0 "es par"
> else
> print $0 "es impar" }'
```

En este ejemplo, si la expresión `$0%2==0` es cierta (es decir, el valor que se le pasa es divisible entre 2), entonces se ejecuta la primera sentencia *print*, sino se ejecuta la segunda sentencia print.
Si el else aparece en la misma línea que el *cuerpo-then*, y el *cuerpo-then* no es una sentencia compuesta (no aparece entre llaves) entonces un punto y coma debe separar el *cuerpo- then* del *else*. Para ilustrar esto veamos el siguiente ejemplo.

```
$ awk '{ if ($0 % 2 == 0) print $0 "es par"; else print $0 "x es impar"}'
```

Si se olvida el ´provocará un error de sintaxis. Se recomienda no escribir la sentencia *else* de esta forma, ya que puede llevar al lector a confusión.

6. ANEXO II. Consulta de información de sistema en Mac OS X.

A continuación, se describe brevemente como se puede consultar la información del sistema en Mac OS X. Además de las aplicaciones Información del Sistema y Monitor de Actividad que proporcionan gráficamente toda la información en tiempo real de los procesos y del sistema, también se puede consultar la información por línea de comandos.

En LINUX la información de la CPU se encuentra en el archivo `/proc/cpuinfo`. Por ejemplo, para mostrar el nombre de de la CPU en Linux se hace:

```
$cat /proc/cpuinfo | grep "model name"
```

En OSX, se utiliza el comando **sysctl**, para consultar determinados aspectos del estado del kernel de OSX. Por ejemplo, para obtener el nombre en OSX se haría:

```
$sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
```

Y para obtener la información completa en OSX:

```
$sysctl -a | grep machdep.cpu
```

Si desea obtener el número de núcleos e hilos, se puede utilizar las siguientes ordenes:

```
$sysctl -a | grep machdep.cpu | grep core_count
$sysctl -a | grep machdep.cpu | grep thread_count
```