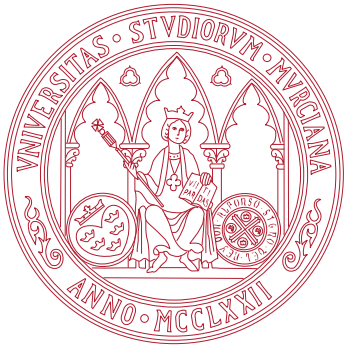

Universidad de Murcia

Facultad de Informática

Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

TÍTULO DE GRADO EN
CIENCIA E INGENIERÍA DE DATOS

Fundamentos de Computadores

Práctica 6: Monitorización del sistema en Linux

Boletines de prácticas

CURSO 2022 / 23

Índice general

Boletines de prácticas

B6.1. Boletín 1: Monitorización del sistema en Linux

B6.1.1. Objetivos

Este boletín pretende mostrar al alumno algunos de los conceptos clave relacionados con las abstracciones y servicios ofrecidos por el sistema operativo (SO). Gracias a la información que Linux pone a disposición del usuario, el alumno podrá observar de primera mano aspectos acerca del hardware disponible, el estado de los procesos, su espacio de direcciones virtual, la ocurrencia de los cambios de contexto, las llamadas al sistema, incluyendo el papel de la biblioteca de sistema, etc. Se verá también cómo los procesos manejan los ficheros abiertos, introduciendo el concepto de descriptor de fichero, y finalmente se mostrará cómo grado de uso de los dispositivos de E/S afecta al número de interrupciones que reciben las CPUs.

B6.1.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura del boletín por parte del alumno.
2. Seguimiento de ejemplos presentados por el profesor.
3. Realización de los ejercicios propuestos en el boletín, con supervisión del profesor, y completándolos en su caso como trabajo autónomo.

B6.1.3. El sistema de ficheros virtual `/proc`

El kernel de Linux tiene dos funciones principales: controlar el acceso a los dispositivos físicos del ordenador (CPU, memoria y entrada/salida) y planificar cuándo y cómo los procesos interactúan con estos dispositivos. El sistema de ficheros virtual `/proc` contiene una jerarquía de ficheros especiales que representan el estado actual del kernel, y permite a aplicaciones y usuarios asomarse a la visión que el kernel tiene del sistema.

En Linux, todos los datos se almacenan como ficheros. La mayoría de los usuarios están familiarizados con los dos tipos principales de ficheros: de texto y binarios. Pero el directorio `/proc` contiene otro tipo de fichero llamado fichero virtual, que no existe físicamente en disco, sino que el núcleo lo crea en memoria, con el fin de ofrecer información acerca del hardware del sistema y los procesos que se están ejecutando. La mayoría de ficheros en el sistema de ficheros virtual `/proc` tienen un tamaño de cero bytes, a pesar de lo cual al mostrarlos con `less` podemos encontrar una gran cantidad de información. Además, la mayoría de estos ficheros tiene como fecha y hora de modificación la actual, lo que indica que se actualizan constantemente. Además, algunos de los ficheros que contiene `/proc` pueden ser manipulados para comunicar cambios de configuración al kernel.

Algunos de los ficheros virtuales que contiene el directorio `/proc` en su nivel más externo proporcionan una **visión actualizada del hardware y del sistema operativo**. En esta sección veremos en más detalle los ficheros `/proc/cpuinfo` y `/proc/meminfo`, mientras que más adelante en el boletín veremos `/proc/interrupts`, que registra las interrupciones generadas por los dispositivos hardware que ha recibido cada CPU. Otros ficheros interesantes son `/proc/vmstat`, que muestra estadísticas acerca del estado de la memoria virtual; y `/proc/version`, que contiene la versión exacta del kernel que se está ejecutando.

Por otro lado, el directorio `/proc` contiene **un subdirectorio por cada proceso en ejecución en el sistema**. Cada uno de estos directorios tiene por nombre el entero que actúa como identificador de proceso (PID), `/proc/<PID>`.

`/proc/cpuinfo`

Muestra información acerca del procesador tal como su marca, modelo, número de núcleos, tamaño de caché, como puede verse en el siguiente ejemplo:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
[...]
cpu MHz         : 800.110
cache size      : 12288 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
[...]
flags           : fpu vme [...] mmx [...] sse sse2 [...] avx2 [...]  rtm [...]
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf  [...]
[...]
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual

processor       : 11
[...]
physical id     : 0
core id         : 5
[...]
```

Como es lógico, los valores mostrados variarán en función de la máquina que se esté usando; en el ejemplo mostrado, vemos que es una CPU Intel Core i7 de 8ª generación (*model name*) con 6 núcleos (*cpu cores*) y 12MB de caché (*cache size*). Fijándonos en el número de "hermanos" (*siblings*), vemos que es el doble que el número de núcleos (12), lo cual nos indica que la CPU tiene soporte SMT (*simultaneous multithreading*, o en el argot de Intel, *hyperthreading*) y que dicha característica está activada. Por tanto, como cada uno de los 6 núcleos de procesamiento es capaz de ejecutar dos hilos a la vez (es decir, ejecutar instrucciones procedentes de dos flujos distintos), a nivel lógico el SO percibe la existencia de 12 procesadores. Por brevedad, se muestra únicamente (un extracto de) la información correspondiente al procesador 0, pero esta se repite para cada uno de los 12 procesadores lógicos. Otros aspectos interesantes que podemos ver es el tamaño del bloque de caché, de 64 bytes (*cache_alignment*), los tamaños de la dirección de memoria física y virtual (39 y 48 bits, respectivamente), las extensiones al repertorio de instrucciones x86-64 soportadas (*flags*) o los fallos de seguridad (*bugs*) a los que este modelo procesador es vulnerable. Nótese que todos los procesadores lógicos comparten el mismo *physical id*, pues el ejemplo corresponde a una máquina *single socket* en la que existe un único chip físico para la CPU que integra todos los núcleos.

`/proc/meminfo`

Muestra información sobre la memoria física (tamaño total, utilización, etc.), como por ejemplo:

```
MemTotal:        3878908 kB
MemFree:          251512 kB
MemAvailable:     718028 kB
```

En las primeras líneas del fichero tenemos un resumen de la información básica sobre la memoria que normalmente necesitamos, como cuánta RAM tiene nuestro sistema o cuánta está disponible en el momento actual: *MemTotal* muestra el total de memoria RAM utilizable (que se deriva del tamaño de la RAM física instalada menos algunos bits reservados y el código binario del kernel); *MemFree*, el total de RAM libre; *MemAvailable* es una estimación de cuánta memoria está disponible para ser reservada cualquier proceso.

`/proc/<PID>/`: Directorio de un proceso en ejecución

En cada uno de estos directorios tenemos a nuestro alcance una gran cantidad de información sobre el proceso en cuestión, entre la que cabe destacar los siguientes ficheros y directorios:

- `cmdline` - Contiene el comando emitido al iniciar el proceso.
- `cwd` - Un enlace simbólico al directorio de trabajo actual del proceso.
- `exe` - Un enlace simbólico al fichero ejecutable del programa ejecutado por este proceso.
- `fd` - Un directorio que contiene todos los descriptores de fichero para este proceso. Los descriptores se muestran mediante enlaces numerados.
- `maps` - Una lista que muestra el uso del espacio de direcciones virtual del proceso, incluyendo las diferentes áreas de memoria (código, datos, pila, bibliotecas) así como el mapeo o correspondencia entre dichas áreas de memoria y los ficheros (ejecutables, bibliotecas) asociados con este proceso. En la siguiente sección veremos esta información con más detalle.
- `status` - El estado del proceso (durmiendo, ejecutando, etc.), incluyendo uso de memoria. En este fichero también se muestran los cambios de contexto experimentados por el proceso.

B6.1.4. Monitorizando el espacio de direcciones virtual de un proceso: `/proc/<PID>/maps`

En la sección anterior vimos que podemos acceder al fichero `maps` del directorio de un proceso para conocer su mapa de memoria. También es posible mostrar dicho mapa de memoria mediante el comando `pmap`, el cual muestra en un formato más legible las áreas de memoria del proceso cuyo PID recibe como parámetro. Por ejemplo, si ejecutamos el comando `sleep 1000 &` y a continuación vemos su mapa de memoria con `pmap -px $(pgrep sleep)`, veremos un contenido similar al siguiente:

```
109458:   sleep 1000
Dirección        Kbytes     RSS   Sucio Modo  Asignaciones
0000564906a46000       8       8       0 r---- /usr/bin/sleep
0000564906a48000      16      16       0 r-x-- /usr/bin/sleep
0000564906a4c000       4       4       0 r---- /usr/bin/sleep
0000564906a4e000       4       4       4 r---- /usr/bin/sleep
0000564906a4f000       4       4       4 rw--- /usr/bin/sleep
[...]
00007fbe9fa52000     160     160       0 r---- /usr/lib/x86_64-linux-gnu/libc.so.6
00007fbe9fa7a000    1620     984       0 r-x-- /usr/lib/x86_64-linux-gnu/libc.so.6
00007fbe9fc0f000     352     192       0 r---- /usr/lib/x86_64-linux-gnu/libc.so.6
00007fbe9fc67000      16      16      16 r---- /usr/lib/x86_64-linux-gnu/libc.so.6
00007fbe9fc6b000       8       8       8 rw--- /usr/lib/x86_64-linux-gnu/libc.so.6
[...]
00007ffc214be000     132      16      16 rw---   [ pila ]
[...]
---------------- ------- ------- -------
```

En cada una de las filas tenemos un área de memoria, para cada cual se muestra en columnas diversa información tal como su dirección de comienzo, tamaño, modo (permisos) o el fichero en disco asignado a dicho área de memoria (el fichero que determina el contenido inicial de dicho área en memoria). Por ejemplo, las cinco primeras áreas corresponden al propio fichero ejecutable del programa, ubicado en la ruta `/usr/bin/sleep`. Fíjate que la segunda área tiene permisos de lectura y ejecución (`r-x--`), lo cual nos da una idea de que en esas direcciones está el código (instrucciones) del programa. De igual forma, la quinta área, cuyo modo es *lectura y escritura* (`rw---`) contiene parte de los datos que utiliza el programa, en concreto, las variables globales. También podemos ver que hay otras cinco áreas de memoria con permisos similares a las anteriores, que corresponden al fichero de la biblioteca del sistema `/usr/lib/x86_64-linux-gnu/libc.so.6`. En estas áreas se ubica el código y los datos de funciones de la biblioteca estándar de C (*libc*) tales como `printf`, que son enlazados dinámicamente en el momento de lanzar a ejecución el programa `sleep`. Existe otro área de memoria con permisos de lectura y escritura cuyo contenido no está asignado a ningún fichero: la *pila* del proceso. Esta zona de memoria es esencial para el soporte de procedimientos, ya que en ella se guardan, en orden LIFO (*last-in-first-out*) valores tales como los parámetros pasados

a los procedimientos o las direcciones de retorno al invocador, las variables locales de cada procedimiento en la pila de llamadas actual, etc.

Las direcciones de memoria donde comienza cada área (primera columna) variarán con toda seguridad cuando reproduzcas este paso en tu máquina, así como entre sucesivas ejecuciones del proceso: compruébalo terminando el proceso con `pkill sleep`, ejecutándolo nuevamente y visualizando su mapa de memoria. Esto se debe a que, por defecto, el kernel utiliza una técnica llamada *aleatorización en la disposición del espacio de direcciones*, que se introdujo como medida de seguridad en el kernel de Linux para proteger frente a ataques por desbordamiento de búfer. Si muestras el contenido del fichero virtual `/proc/sys/kernel/randomize_va_space`, verás el valor 2, que indica que se está usando aleatorización completa. En caso de que estés en una máquina sobre la que tengas permiso de administrador, puedes probar a desactivar temporalmente esta técnica escribiendo un 0 en dicho fichero:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

A continuación, prueba a ejecutar el comando `sleep` repetidamente, mostrando su mapa de memoria, y verás que cada nuevo proceso lanzado a partir de ese momento tiene las mismas áreas de memoria en las mismas direcciones del espacio virtual que el proceso anterior.

B6.1.5. Monitorizando las llamadas al sistema efectuadas por un proceso

Cada interacción con el mundo exterior está mediada por el núcleo a través de llamadas al sistema. Si una aplicación guarda un archivo, escribe en el terminal o abre una conexión TCP, el núcleo está involucrado. Estas llamadas al sistema son llamadas de función desde una aplicación al kernel, que utilizan un mecanismo específico por razones de seguridad (pues el código de usuario no sabe –ni debe saber– dónde se ubica el código del kernel), pero que en realidad no son más que una llamada a la API del kernel. A menudo nos referimos a las llamadas al sistema mediante el término inglés *syscall*, para abreviar. En esta sección veremos en qué se diferencian las llamadas al sistema de las llamadas a una biblioteca, y las herramientas que permiten hurgar en esta interfaz SO/aplicación.

A continuación se muestra el código C de un sencillo programa, `pid.c`, que simplemente recupera su id de proceso a través de `getpid`. En Linux, un proceso no nace conociendo su PID, sino que debe pedírselo al kernel, por lo que esto requiere una llamada al sistema:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t p = getpid();
    printf("%d\n", p);
}
```

El programa realiza una llamada a la función `getpid` de la biblioteca C, una función *wrapper* que es la que realmente realiza la llamada al sistema. En realidad, cuando llamas a funciones como `open`, `read` y similares, estás llamando a estas funciones "envoltura" de la biblioteca del sistema. Estas funciones *wrappers* ofrecen comodidad sobre la API básica del SO, ayudando a mantener el kernel reducido (recuerda que el código del kernel se ejecuta en modo privilegiado, donde los errores pueden ser desastrosos). Todo lo que se pueda hacer en modo usuario debería hacerse en modo usuario. Por tanto, las bibliotecas ofrecen métodos amigables, incluyendo procesamiento exhaustivo de argumentos, incluso hacen algo de *caching*: esto es lo que hace `getpid()` de libc: cuando se llama por primera vez, realmente realiza una llamada al sistema, pero el PID se almacena en caché para evitar la sobrecarga de la llamada al sistema en las siguientes invocaciones.

La herramienta `strace` resulta de gran utilidad a la hora de ilustrar las interacciones entre los procesos y el kernel de Linux. `strace` es una utilidad de diagnóstico mediante la que podemos observar, entre otra información, las llamadas al sistema que un determinado proceso realiza para solicitar servicios al kernel tales como crear o terminar un proceso, leer o escribir en un fichero, etc. En el caso más simple, `strace` ejecuta el comando especificado hasta que sale, interceptando y registrando las llamadas al sistema que son invocadas por el proceso, así como las señales recibidas. El nombre de cada llamada al sistema, sus argumentos y su valor de retorno se imprimen en la salida de error o en el fichero especificado con la opción `-o`. Es posible indicar qué llamadas al sistema concretas se desea

tracear con la opción `--trace=...`. Así, por ejemplo, con el siguiente comando podemos tracear las llamadas al sistema invocadas durante la ejecución del programa anterior (`pid.c`):

```
strace --trace=execve,getpid,write -o strace-pid.log ./pid
```

Podemos ver la traza de llamadas al sistema monitorizadas volcando el contenido del fichero `strace-pid.log`:

```
execve("./pid", ["./pid"], 0x7ffc2b969228 /* 50 vars */) = 0
getpid()                                = 111442
write(1, "111442\n", 7)                 = 7
```

Como vemos, en primer lugar el proceso hijo creado por el *shell* utiliza la *syscall* `execve` para ejecutar el programa `pid`, reemplazando el que estaba ejecutado hasta ese instante (`bash`). Una vez se está ejecutando `pid`, vemos cómo la llamada al sistema `getpid` devuelve el PID del proceso (111442). Después, el código de la función `printf` de la librería de C convierte dicho valor entero a una cadena de caracteres, que después imprime por la salida estándar (pantalla) mediante la llamada al sistema `write`; esta *syscall* recibe el descriptor del fichero en el que escribir (el descriptor 1 indica que se trata de la salida estándar, i.e., la pantalla), la dirección de memoria donde comienza la cadena de caracteres a imprimir y su longitud (7 en total, los 6 dígitos del PID más el carácter de nueva línea, '`\n`'). Finalmente, a su regreso del kernel `write` devuelve el número de bytes escritos en el fichero.

B6.1.6. Monitorizando los ficheros abiertos por un proceso: `/proc/<PID>/fd`

En el subdirectorio `fd` de cada proceso en `/proc` podemos observar enlaces simbólicos numerados que apuntan a los ficheros abiertos por dicho proceso en ese instante. Cada número corresponde al descriptor de fichero: un número entero no negativo, que el kernel devuelve como resultado de la llamada al sistema `open` (a la cual se le pasa la ruta al fichero), y que el proceso utiliza desde ese momento para identificar dicho fichero abierto en las sucesivas llamadas al sistema (`read`, `write`, etc.) para acceder a dicho fichero. Todos los procesos lanzados desde el shell tiene disponibles los descriptores de fichero 0, 1 y 2, que corresponden, respectivamente, a la entrada estándar, salida estándar y salida de error del proceso. Podemos confirmar este punto ejecutando la siguiente secuencia de comandos:

```
tail -f ~/.bash_history & ls -l /proc/$(pgrep tail)/fd; pkill tail
```

Vemos que primero se ejecuta `tail` en segundo plano (para mostrar el final del fichero que guarda el historial de comandos) y seguidamente se hace un listado largo del directorio `fd` del proceso creado, utilizando un *subshell* para averiguar con `pgrep` el PID de dicho proceso. La salida resultante podría ser similar a esta:

```
[...]
lrwx------ 1 rtitos rtitos 64 oct 18 10:16 0 -> /dev/pts/0
lrwx------ 1 rtitos rtitos 64 oct 18 10:16 1 -> /dev/pts/0
lrwx------ 1 rtitos rtitos 64 oct 18 10:16 2 -> /dev/pts/0
lr-x------ 1 rtitos rtitos 64 oct 18 10:16 3 -> /home/rtitos/.bash_history
lr-x------ 1 rtitos rtitos 64 oct 18 10:16 4 -> anon_inode:inotify
```

Como vemos, los ficheros se muestran como enlaces simbólicos, como indica el carácter 'l' (*link*) en la primera columna. Los tres primeros descriptores (0, 1 y 2) apuntan al fichero `/dev/pts/0` que se corresponde con un dispositivo de E/S, en este caso el terminal (teclado+pantalla) desde el que se ha lanzado el comando `tail`. A través de este dispositivo, representado como un fichero en el directorio `/dev`, el proceso podrá leer datos de teclado (*stdin*) y mostrar mensajes por pantalla (*stdout* y *stderr*) de la misma forma que en que lo haría si se tratase de un fichero regular en un disco. En este caso concreto, `/dev/pts/0` es un *pseudoterminal* creado por el programa emulador de terminales `gnome-terminal` (cada pestaña corresponde a un pseudoterminal con un número distinto). Tras los tres enlaces que apuntan al terminal, vemos que el siguiente enlace simbólico (número 3) apunta al fichero `.bash_history`, lo cual nos indica que el programa `tail` ha obtenido dicho descriptor al abrir dicho fichero para poder mostrar sus últimas líneas, y todavía no ha sido cerrado en el instante en que se ejecuta `ls`.

Si probamos a visualizar el contenido del directorio `fd` de procesos que estén ejecutando programas más complejos, tal como un navegador web, veremos que el número de descriptores de fichero puede llegar a ser muy grande. Por otro lado, es probable que junto a descriptores asociados a ficheros regulares en el sistema de ficheros, veamos otros descriptores que correspoden a ficheros de tipo *pipe* y de tipo *socket*, mecanismos ofrecidos por el kernel para la comunicación con otros procesos, que en el caso de los *sockets* pueden encontrarse en una máquina remota.

Por último, para monitorizar los ficheros abiertos en cada momento podemos utilizar el comando `lsof` (*list open files*). Invocado sin parámetros, `lsof`, muestra una lista de todos ficheros abiertos por todos los procesos en ejecución. Si se le pasa como parámetro una ruta a un fichero, mostrará únicamente los procesos que están accediendo al mismo.

B6.1.7. Monitorizando las interrupciones hardware mediante `/proc/interrupts`

En el fichero `/proc/interrupts` el kernel registra las distintas interrupciones recibidas en cada una de las CPUs. Nos centraremos exclusivamente en aquellas interrupciones hardware generadas por los dispositivos de E/S. El contenido que observamos en dicho fichero variará en cierta medida de un sistema a otro, ya que depende tanto del número de CPUs como de los dispositivos con los que cuenta el computador. Así pues, en la máquina virtual proporcionada para el desarrollo de las prácticas de esta asignatura, las primeras líneas del fichero `/proc/interrupts` serán similares a estas:

```
           CPU0
  0:         33   IO-APIC   2-edge      timer
  1:        558   IO-APIC   1-edge      i8042
  8:          0   IO-APIC   8-edge      rtc0
  9:          0   IO-APIC   9-fasteoi   acpi
 12:        394   IO-APIC  12-edge      i8042
 14:        737   IO-APIC  14-edge      ata_piix
 15:        729   IO-APIC  15-edge      ata_piix
 18:       1897   IO-APIC  18-fasteoi   vmwgfx
 19:     111306   IO-APIC  19-fasteoi   enp0s3
 20:       4175   IO-APIC  20-fasteoi   vboxguest
[...]
```

Para observar cómo cambia el número de interrupciones recibidas a lo largo del tiempo, utilizaremos el comando `watch`, el cual ejecuta periódicamente el comando pasado como parámetro:

```
watch -d cat /proc/interrupts
```

La opción `-d` se utiliza para resaltar las diferencias entre actualizaciones sucesivas. Al ejecutarlo de esta forma, es posible comprobar cómo el grado de uso de un dispositivo de E/S afecta directamente al número de veces que éste interrumpe a la CPU. Para mostrar esto con un ejemplo, vamos a monitorizar las interrupciones generadas por el controlador de la interfaz de red (NIC) cuando se descarga un fichero de Internet, comparándolas con las que se producen en un periodo de inactividad en el uso de la red. Para facilitar la visualización, podemos filtrar con `grep` el contenido de `/proc/interrupts` para quedarnos únicamente con la línea correspondiente a las interrupciones del NIC, que en el caso de la máquina virtual se llama `enp0s3`. Para averiguar el nombre exacto de la interfaz de red conectada a Internet podemos utilizar el comando `ifconfig`[1]:

```
alumno@alumno-VirtualBox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
[...]
alumno@alumno-VirtualBox:~$ grep enp0s3 /proc/interrupts
 19:     103062   IO-APIC  19-fasteoi   enp0s3
alumno@alumno-VirtualBox:~$ wget https://ditec.um.es/~rtitos/docencia/fc-gcid/video.mp4
--2022-10-19 13:21:17--  https://ditec.um.es/~rtitos/docencia/fc-gcid/video.mp4
[...]
video.mp4.2         100%[===================>]  96,33M  11,2MB/s    en 8,6s
2022-10-19 13:21:26 (11,2 MB/s) - 'video.mp4.2' guardado [101014694/101014694]
alumno@alumno-VirtualBox:~$ grep enp0s3 /proc/interrupts
 19:     204771   IO-APIC  19-fasteoi   enp0s3
```

En la salida anterior vemos que se producen unas 100.000 interrupciones como consecuencia de la descarga de un fichero de algo menos de 100MB, cuyo tiempo es inferior 10 segundos. Sin embargo, si mostramos dicha línea de `/proc/interrupts` periódicamente, en un escenario donde no estamos ejecutando ningún programa que haga uso de la red (navegador web, etc.), veremos que el número de interrupciones recibidas en el mismo intervalo de tiempo es prácticamente despreciable en comparación.

[1] Puede que necesites instalarlo con `sudo apt install net-tools`

B6.1.8. Ejercicios a realizar durante la sesión

NOTA IMPORTANTE:
Antes de realizar los ejercicios, asegúrate de grabar la sesión con `script -a typescript_prac6_bol1`. Debes empezar cada ejercicio escribiendo el comentario de documentación `### EJERCICIO N ###` . Finalmente, no olvides añadir dicho fichero a tu repositorio-bitácora, respetando la organización de la misma en directorios y subdirectorios.
Recuerda responder a cada pregunta tecleando tu respuesta en el mismo terminal (a continuación de los comandos necesarios para cada apartado) precedida por el carácter #, que indica al *shell* que se trata de un comentario (el shell ignora los siguientes caracteres hasta el siguiente final de línea, cuando pulses INTRO). Puedes introducir tu respuesta en mútiples líneas si así lo deseas, empezando cada línea con #.

1. **Monitorización del hardware.** Accede al fichero oportuno en `/proc` y mediante `grep`, filtra las líneas relevantes y responde a las siguientes preguntas sobre tu sistema:

   a) ¿Cuántos procesadores hay?
   b) ¿Cuál es el tamaño del bloque de caché?
   c) ¿Cuál es el tamaño de la dirección física?
   d) ¿Cuánta memoria hay en total en tu sistema?

2. **Espacio de direcciones virtual de un proceso: Áreas.**

   a) Ejecuta el programa `vaddress` [2], el cual muestra en primer lugar su PID seguido de la dirección de diferentes variables y funciones de dicho programa. Cuando el programa pause su ejecución en espera de pulsar `INTRO`, presiona `CTRL-Z` para detener (dormir) el proceso.
   b) A continuación, muestra el fichero `maps` situado en el directorio correspondiente a este proceso en `/proc/PID/maps`.
   c) Muestra el mapa de memoria de dicho proceso, pero ahora mediante el comando `pmap`. A este comando necesitas pasarle el PID del proceso cuyo mapa de memoria quieres mostrar. *PISTA: puedes obtener el PID con el comando pgrep (pasándole el nombre del programa ejecutado por el proceso que buscas). Prueba a pasar el PID devuelto por pgrep, como parámetro a pmap, usando un subshell.*
   d) ¿En qué área de memoria se ubica `f`? ¿Qué permisos tiene el proceso sobre dicho área de memoria?
   e) ¿En qué área de memoria se ubica `g`? ¿Qué permisos tiene el proceso sobre dicho área de memoria?
   f) ¿En qué área de memoria se ubica `l`? ¿Qué permisos tiene el proceso sobre dicho área de memoria?
   g) ¿En qué área de memoria se ubica `p`? ¿Qué permisos tiene el proceso sobre dicho área de memoria? ¿Qué fichero respalda el contenido de este área de memoria?
   h) En el caso de `f` y `g`, ¿con qué fichero en disco se corresponden sus respectivas áreas de memoria? ¿Qué crees que contiene dicho fichero?
   i) En el caso de `p`, ¿con qué fichero en disco se corresponde su área de memoria? ¿Qué crees que contiene dicho fichero?
   j) Reanuda mediante `fg` la ejecución del proceso pausado anteriormente. Una vez devuelto a ejecución en primer plano, presiona `CTRL-C` para terminar su ejecución.

3. **Código y datos en la memoria de un proceso. Protección de memoria**

   a) Vuelve a ejecutar el programa `vaddress`, y ahora, tras continuar su ejecución pulsando `INTRO`, introduce la dirección de `f` (copia y pega para mayor comodidad) y elige la opción 'x' para ejecutar el contenido de dicha dirección. A la vista del resultado, ¿qué crees que hay en esa dirección de memoria?

[2] Todos los programas necesarios para la realización de los ejercicios están disponibles a través del Aula Virtual.

b) A la vista del código fuente del programa en C `vaddress.c` (líneas 5-25), ¿a qué funciones del programa corresponden, respectivamente, las direcciones `f`, `i` y `p`?

c) ¿Por qué la función `p` se almacena en una zona de la memoria distinta a la de las funciones `f` e `i`?

d) Lanza de nuevo el programa y ahora trata de leer de la dirección de `f`. Observa el resultado y comprueba que se trata de la instrucción `endbr64` del ISA x86-64[3]

e) Lanza de nuevo el programa y ahora trata de escribir en la dirección de `f`. ¿Qué ocurre? Razona la respuesta, relacionando la misma con la información recabada mediante `pmap` en el ejercicio anterior.

f) Lanza de nuevo el programa y ahora trata de escribir en la dirección de `g`. ¿Qué ocurre? Razona la respuesta, relacionando la misma con la información recabada mediante `pmap` en el ejercicio anterior.

g) Lanza de nuevo el programa y ahora trata de leer en la dirección de `g`. Observa el resultado e indica, sabiendo que dichos bytes codifican un entero con signo (32 bits), el valor decimal al que corresponden.

h) A la vista del código fuente del programa en C `vaddress.c` (líneas 5-25), ¿a qué variable del programa corresponde la dirección `g`?

i) Lanza de nuevo el programa y ahora trata de ejecutar el contenido de la dirección de `g`. ¿Qué ocurre? Razona la respuesta, relacionando la misma con la información recabada mediante `pmap` en el ejercicio anterior.

j) Lanza de nuevo el programa y ahora trata de ejecutar en la dirección de `i`. ¿Qué ocurre? Razona la respuesta.

k) Lanza de nuevo el programa y ahora trata de escribir en la dirección de `l`. ¿Qué ocurre? Razona la respuesta.

l) A la vista del código fuente del programa en C `vaddress.c` (líneas 5-25), ¿a qué variable del programa corresponde la dirección `l`? ¿Por qué dicha variable se almacena en una zona de la memoria distinta a la de `g`?

m) Por último, lanza de nuevo el programa y ahora trata de leer en una dirección que no pertenezca a ninguna de las áreas mostradas (p.ej. la dirección 0). ¿Qué ocurre? Razona la respuesta.

4. **Monitorizando el estado de un proceso. Cambios de contexto**

a) Ejecuta los programas `idle` y `busy`, lanzando ambos en segundo plano.

b) A continuación, muestra el estado de cada uno de los dos procesos accediendo al fichero virtual `/proc/PID/status`. Puedes filtrar con el comando:
`grep State /proc/{pid1,pid2}/status`
para quedarte únicamente con el estado de cada proceso. Usando el historial de comandos, repite este comando varias veces, observando el estado de cada uno.

c) Repite el apartado anterior, pero en esta ocasión filtrando para quedarte con el número de cambios de contexto (filtra esta vez por la cadena `switches`).

d) A la vista de la información observada en los apartados anteriores, y del código fuente de ambos programas (`idle.c`, `busy.c`), explica razonadamente el comportamiento de cada proceso, comparando y justificando las diferencias que veas en el estado y el número de cambios de contexto voluntarios e involuntarios que experimenta.

e) Por último, termina ambos procesos lanzados en segundo plano utilizando el comando `pkill`.

5. **Monitorización de las llamadas al sistema. Entrada y salida estándar y de error**

a) Ejecuta el programa `syscalls`, cuyo código fuente se encuentra en el fichero `syscalls.c`, y observa su funcionamiento. ¿Qué es lo hace este programa?

[3] `https://www.felixcloutier.com/x86/endbr64`

b) Lanza ahora el programa con `strace` para tracear las llamadas al sistema.
`strace ./syscalls`
Date cuenta de que, por defecto, `strace` envía su salida a la pantalla, donde se mezcla con la salida del propio comando que está siendo traceado.

c) Repite el apartado anterior, pero ahora pasando la opción `-o fichero_traza` para enviar la salida de `strace` a un fichero (NOTA: las opciones de `strace` deben ir *antes* que el parámetro (comando a *tracear*). Al terminar la ejecución, muestra el contenido del fichero generado con la traza de llamadas al sistema.

d) Repite el apartado anterior, pero ahora pasando además la opción `--trace=read,write,execve` para tracear únicamente las llamadas relacionadas con la lectura y escritura en ficheros (*read,write*), y con la ejecución de un programa a partir de un fichero ejecutable (*execve*):
`strace --trace=execve,read,write -o output_syscalls.strace ./syscalls`

e) Numera con `nl` las líneas del fichero generado con la traza de llamadas al sistema (`output_syscalls.strace`). A partir de dicha información, responde a la siguientes preguntas.

f) Indica la(s) línea(s) en la(s) que se realiza una llamada al sistema para escribir por la salida estándar

g) Indica la(s) línea(s) en la(s) que se realiza una llamada al sistema para escribir por la salida de error.

h) Indica la(s) línea(s) en la(s) que se realiza una llamada al sistema para leer de la entrada estándar.

i) En el momento de la primera llamada a `execve`, ¿el código de qué programa se está ejecutando?

j) En el momento de la última llamada a `write`, ¿el código de qué programa se está ejecutando?

6. **Monitorización de las llamadas al sistema. Manejo de ficheros.**

a) Ejecuta el programa `fileio` y observa su funcionamiento. El programa crea un fichero nuevo (o si ya existe, trunca su contenido para dejarlo vacío), a continuación escribe una cadena de caracteres en el mismo y finalmente lo cierra. ¿Cuál es el nombre y el contenido del fichero creado?

b) Al igual que en el ejercicio anterior, usa `strace` para monitorizar las llamadas al sistema que lleva a cabo el programa `fileio`, con el siguiente comando:
`strace --trace=read,write,openat,close -o output_fileio.strace ./fileio`
A partir de la información en `output_fileio.strace`, responde a la siguientes preguntas.

c) ¿Cuál es el descriptor de fichero usado para escribir en el fichero creado? ¿Qué llamada al sistema devolvió dicho descriptor de fichero?

d) ¿Qué valor devuelve la llamada al sistema `openat` en caso de error?

e) ¿Qué valor devuelve la llamada al sistema `close` en caso de éxito?

f) A la vista del código fuente del programa en C `fileio.c`, indica desde qué función de la biblioteca estándar de C de las utilizadas (`fprintf`, `fopen`, etc.) se invoca cada una de las llamadas al sistema traceadas en este ejercicio.

7. **Monitorización de los ficheros abiertos.**

a) Ejecuta el siguiente comando para generar un fichero de texto con 20 líneas:
`for i in $(seq -w 1 20) ; do echo "Line $i"; done > input_openfiles.txt`
Comprueba que ha sido generado correctamente mostrado su contenido con `cat`.

b) Ejecuta el programa `openfiles`, que lee del fichero `input_openfiles.txt` (el cual debes haber creado previamente en el mismo directorio desde donde lances el programa `openfiles`) y a partir de los datos leídos genera un nuevo fichero `output_openfiles`.

c) A la vista del contenido de `output_openfiles`, explica qué tarea realiza el programa `openfiles` con respecto a los datos del fichero de entrada.

d) Ejecuta de nuevo el programa. Cuando el programa pause su ejecución en espera de pulsar `INTRO`, presiona `CTRL-Z` para detener (dormir) el proceso.

e) Averigua el PID del proceso detenido mediante `pgrep openfiles` y a continuación haz un listado largo `ls -l` del directorio `/proc/<PID>/fd`.

f) A la vista de los ficheros abiertos por el proceso en ese instante, así como del código fuente del programa `openfiles.c` (bucle a partir de la línea 44), indica en qué orden crees que el programa recupera los datos del fichero de entrada.

g) Ejecuta el comando `lsof input_openfiles.txt` para ver los procesos activos que tienen abierto dicho fichero.

h) El comando `lsof` muestra todos los ficheros abiertos por los procesos activos en el sistema. Filtra la salida de dicho comando con `grep` para mostrar únicamente los ficheros abiertos por el proceso `openfiles`.

i) Además de los ficheros de datos de entrada y salida, ¿qué otros ficheros regulares tiene abiertos dicho proceso? ¿Para qué crees que se usan estos ficheros en la ejecución del programa?

j) Muestra mediante `lsof` todos los procesos activos que están haciendo uso del fichero que contiene la biblioteca estándar de C (p.ej., `/lib/x86_64-linux-gnu/libc.so.6`).

k) A continuación, teclea `fg` para volver poner en primer plano el proceso detenido, y pulsa `INTRO` para terminar su ejecución.

8. **Monitorizando las interrupciones de los dispositivos.** Para realizar este apartado, es aconsejable que no estés ejecutando en la máquina virtual (o nativa) ningún otro programa más que el terminal del *shell*.

a) Identifica con `lspci -v` el dispositivo correspondiente al controlador de disco SATA, y averigua el número de su señal de petición de interrupción (IRQ, *interrupt request*), que se muestra en la primera columna de cada línea del fichero `/proc/interrupts`.

b) Una vez hayas localizado la línea correspondiente al controlador de disco, ejecuta el siguiente comando, sustituyendo `NUM` por el número que has averiguado en el apartado anterior:
```
for i in $(seq 1 10) ; do grep NUM: /proc/interrupts; sleep 1; done
```

c) En promedio, ¿cuántas interrupciones, aproximadamente, se reciben en la CPU en cada segundo?

d) Ahora, ejecuta el siguiente comando para generar, en segundo plano, un fichero archivador con el contenido íntegro del directorio `/usr` (y pasa seguidamente a realizar el siguiente apartado):
```
tar zcf usr.tar.gz /usr &
```

e) Haciendo uso del historial, vuelve a ejecutar el bucle `for` anterior para mostrar las interrupciones durante 10 segundos. Indica si observas alguna diferencia y explica a qué crees que se debe.