



El objetivo de esta actividad es comprobar lo aprendido hasta ahora sobre Llamadas al Sistema, la Tabla de Canales, y sobre las LIS que manejan la información de los inodos.

Recuerda que tienes la descripción de las llamadas al sistema en el tomo 2 de la ayuda man, escribiendo en la línea de comandos `man <llamada>` o buscando en <https://www.kernel.org/doc/man-pages/>

4.1. Concepto de Llamada al Sistema: `strace` y `ltrace`

Prueba este programa con las instrucciones entre las líneas 4 y 9, y comprueba qué funciones de librería usa y qué llamadas al sistema hace.

```

1 #include <stdio.h>
  #include <sys/types.h>
  #include <unistd.h>
4 /**
  $ gcc -no-pie -o nc cuenta.c
  $ ./nc
7 $ echo $?
  $ ltrace ./nc
  $ strace ./nc
10 */
int main(){
  int numcuenta;
13   numcuenta=getuid();
   printf("\nSoy la cuenta %d\n", numcuenta);
   return (numcuenta-745); // Para verlo, echo $? en la línea de comandos
16   // Solamente se puede devolver un valor menor que 256
}

```

Fichero 4.1: cuenta.c

Prueba el siguiente programa con estas instrucciones.

```

1 gcc -no-pie -o compara compara.c
  ./compara
  strace ./compara
4 strace -e trace=open,read,write ./compara
  ltrace ./compara

```

```

1 #include <stdio.h>
  #include <string.h>
int main() {
4   printf("\nVamos a comparar dos cadenas\n");
   char a[]="hOLA";
   char b[]="hola";
7   int n=strcmp(a,b);
   printf("\nLa diferencia es %d \n",n);
}

```

Fichero 4.2: compara.c

Para saber el efecto *puro* del programa, elimina `ltrace`. Comprueba la diferencia con la anterior, ésta da las llamadas a librerías mientras que `strace` informa de las llamadas al sistema.

Observa los siguientes programas:

```

#include <stdio.h>
#include <sys/types.h>
3 #include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
6 int main(){
    FILE * f; // fichero
    f=fopen("hola.txt", "w");
9    fprintf(f, "Hola... C\n");
    fclose(f);
    return 0; // Para verlo, echo $? en la línea de comandos
12 }

```

Fichero 4.3: c-abre.c

```

#include <stdio.h>
#include <sys/types.h>
3 #include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
6 int main(){
    int fd; // entrada en la tabla de canales
    fd=open("hola.txt", O_WRONLY|O_CREAT, 0666);
9    write(fd, "Hola... S\n", 8);
    close(fd);
    return 0; // Para verlo, echo $? en la línea de comandos
12 }

```

Fichero 4.4: s-abre.c

Pruébalos con las siguientes instrucciones. Analiza si dan igual resultado o diferente:

```

gcc -no-pie -o c c-abre.c
./c
3 strace ./c
ltrace ./c

```

```

gcc -no-pie -o s s-abre.c
2 ./s
strace ./s
ltrace ./s

```

Prueba el programa `compara.c` (el primero de esta sección) con estas instrucciones y justifica por qué no se hace ninguna llamada al sistema entre los dos `printf` pero sí que aparece `strcmp` con `ltrace`.

Más sobre `strace`: prueba estas instrucciones. Para saber su efecto *puro*, elimina `strace`.

```

echo Hola # imprime Hola en la pantalla, salida estándar
2 strace echo Hola # visualiza las llamadas al sistema
ls -lR /home/ > salida.txt # listado recursivo de ficheros de
# un directorio, y en vez de
5 # salir en pantalla se escribe en un fichero (redirección)
strace -c ls -lR /home/ > salida.txt # tiempo usado en las llamadas
strace -r sleep 1 # tiempo empleado en cada llamada al sistema

```

Filtrando sólo ciertas llamadas al sistema:

```
strace -e trace=openat,read,write echo Hola
2 strace -e file echo Hola
```

Diferencia entre la salida de los dos comandos:

```
1 diff <(strace -e file echo Hola 2>&1) <(strace echo Hola 2>&1)
```

Observamos cómo hace la línea de comandos la redirección:

```
bash -c 'echo Hola' # ejecutar la instrucción en una nueva línea de comandos
2 strace bash -c 'echo Hola'
strace bash -c 'echo Hola >f1.txt' # cómo hace bash la redirección
```

Más sobre las últimas tres líneas

en DTrace: [even better than] strace for OS X

Un ejemplo de la utilidad de este comando:

Debug like a sysadmin: using strace and ltrace

Para profundizar:

<http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html>

4.2. Enlaces duros y enlaces simbólicos

Observa la información de las instrucciones `stat` y lo que se ve con `ls` al hacer enlaces duros y simbólicos. Observa el tamaño de los ficheros, el número de inodo y el número de enlaces duros. ¿Qué ocurre cuando se borra el fichero `hola.txt` y se crea otro?

```
vi hola.txt
ls -l
3 ls -li *.txt # cada cambio que hagamos lo comprobamos con esta instrucción
ln hola.txt kaixo.txt
stat hola.txt
6 stat kaixo.txt
ls -li *.txt # revisamos los inodos
cat kaixo.txt
9 cat hola.txt
vi kaixo.txt # cambiamos el fichero
cat hola.txt # y se ven los cambios en los dos nombres de fichero
12 ln kaixo.txt hi.txt # tres enlaces duros
vi hi.txt
ln -s kaixo.txt es1.txt # hacemos enlaces simbólicos
15 ln -s hola.txt es2.txt
ls -li *.txt
rm hola.txt # un enlace duro menos
18 vi hola.txt
ls -li \
*.txt # en dos líneas separadas por un salto (intro justo después de la \ )
21 history | tail -25 # los últimos 25 comandos
history | tail -25 >e2.txt # los guardamos en un fichero
```

4.3. Llamadas al Sistema sobre ficheros

En esta parte vamos a comprobar el comportamiento de estas Llamadas al Sistema.

unlink, link

Tenemos las llamadas al sistema `link` y `unlink`. Ver `man 2 unlink` o también <http://man7.org/linux/man-pages/man2/unlink.2.html>

Operaciones sobre enlaces duros:

```
2 strace mv f1.txt f2.txt # comparar con el que haremos con link y unlink
  strace rm f2.txt      # usa unlink
```

A partir de ellas hay que hacer unos programas equivalentes a los comandos:

1. `rm` (borrar)
2. `mv` (mover o renombrar, cambia el camino de acceso a un inodo)
3. `ln` (crea un enlace duro)

Renombrar lo podemos hacer creando otro enlace duro y después borrando el antiguo.

Para ello prueba el programa visto en C que muestra los argumentos:

```
1 #include <stdio.h>
  void main(int argc, char *argv[]) {
    int i;
  4  printf("\nPrograma: %s\n", argv[0]);
    for(i=1; i<argc; i++)
      printf("argumento %d: %s\n", i, argv[i]);
  7 }
```

Si lo llamas `prueba.c` lo ejecutas con:

```
2 gcc -o prueba prueba.c
  ./prueba uno dos tres
```

A partir de este programa, para borrar un fichero deberías crear el fichero `borra.c` y usar la *llamada al sistema* `unlink(argv[1]);` como única instrucción del programa, además de `#include <unistd.h>`, la librería que la contiene.

nice

La `&` ejecuta el proceso en segundo plano, permite escribir más instrucciones.

```
1 sleep 4
  sleep 4 &
```

La segunda instrucción (`sleep 2000`) cuelga de otro proceso.

```
1 nice sleep 1000 &
  bash -c 'sleep 2000' &
  pstree | less
```

Observa cómo actúa `nice` para cambiar la prioridad de un proceso. La `N` indica la menor prioridad. Los corchetes se colocan para que `grep` sólo saque la línea de la ejecución de la instrucción `ps aux`

```
nice ps aux|grep a[u]x
ps aux|grep a[u]x
```

Observa la prioridad absoluta (20 es la normal) y el incremento de 10. Ver <https://www.tecmint.com/set-linux-process-priority-using-nice-and-renice-commands/>

```
1 sleep 1000 &
  nice -n 20 sleep 2000 &
  top
4 sudo nice -n -20 sleep 3000 &
  top
  history | tail -20 > nice.txt
```

Si se usa htop se puede buscar con pulsando / y después sleep.

4.4. Entrada y salida. Canales

Vamos a ver ejemplos sobre la redirección de la E/S y cómo se modifica la Tabla de Canales.

Redirecciones básicas

Prueba las redirecciones básicas. Tienes una explicación detallada en los apuntes: <https://lsi.vc.ehu.eus/pablo/n/docencia/ISO/Apuntes.zip>

```
touch ficheroqueexiste.txt
ls ficheroqueexiste.txt
3 ls noexiste.txt
  ls ficheroqueexiste.txt.txt > c1.txt 2> c2.txt
  cat c1.txt
6 cat c2.txt
  ls noexiste.txt.txt > c1.txt 2> c2.txt
  cat c1.txt
9 cat c2.txt
```

Salida por canales estándares

Salida por los canales 1 y 2 con la sintaxis en C:

```
/* Prueba de distintos canales A */
// gcc canales.c -o c
3 // ./c
// ./c >k1
// ./c 2>k2
6
#include <stdio.h>
#include <stdlib.h>
9 int main(){
  printf("\n1. Canal 1\n"); // Salida estándar
  fprintf(stdout, "\n2. Canal 1\n"); // Salida estándar
12 fprintf(stderr, "\n3. Canal 2\n"); // Salida de errores
  exit(0);
}
```

Observa el siguiente programa `psaluda.c` que le pasamos una cadena y la imprime.

```

#include <stdio.h>
#include <string.h>
3 #include <stdlib.h>

void main(int argc, char * argv[]){
6   char nombre[100];
   if (argc!=2) {
       fprintf(stderr, "\nNº incorrecto de argumentos\n");
9       exit(1);
   }
       // scanf("%s", nombre); // si pedimos la cadena en vez de argv
12 strcpy(nombre, argv[1]);
   if (strcmp(nombre, "")==0) {
       // if (nombre[0]=='\0') // formas alternativas de comprobar
       // if (*nombre==0) // si la cadena está vacía
       // if (!*nombre)
       // if (strlen(nombre)==0)
18   fprintf(stderr, "\nError, tu nombre...\n");
       exit(2);
   }
21 else
       printf("\nTu nombre es %s\n", nombre);
       exit(0);
24 }

```

Fichero 4.6: `psaluda.c`

Pruébalo con estos comandos:

```

vi psaluda.c
gcc -o saluda psaluda.c
3 ./saluda Pablo
echo $?
./saluda a b c d # argc vale 5
6 echo $?
./saluda "Pablo G" # cadena con espacio
echo $?
9 ./saluda "" # le damos una cadena vacía
echo $?

```

Redirecciones

Prueba también las redirecciones con el programa `psaluda.c`.

```

./saluda Pablo > c1.txt 2> c2.txt
2 cat c1.txt
cat c2.txt
./saluda 'Hola, Pablo' >> c1.txt 2> c2.txt
5 ./saluda > c1.txt 2> c2.txt
./saluda Pablo 2>&1 | grep --colour -n -i n
./saluda 2>&1 | grep --colour -n -i n

```

Tabla de Canales

Vamos a comprobar cuál es la tabla de canales de un programa que busca 666 en el generador de números aleatorios, reenvía la salida a un fichero y los errores a otro:

```
grep < /dev/random 66666 2> errores.txt > salida.txt
```

Pulsamos CTRL-Z para parar el programa, y comprobamos en la información del Sistema Operativo (a través de /proc) cuál es la tabla de canales. Para ello averiguamos el PID del proceso con ps.

```
ps
2 ls -l /proc/17165/fd
```

Y podemos terminar el proceso con kill%1 o fg para traerlo al primer plano y luego CTRL-C para pararlo.

Más para fg y bg:

<https://websistent.com/how-to-manage-jobs-in-linux-fg-bg-kill-ctrlz/>

Ejemplo de utilidad de las redirecciones

Si entendemos canales y redirecciones, podemos ver qué hace este código:

```
1 python3 -c \
  'import sys,yaml,json; json.dump(yaml.load(sys.stdin),sys.stdout,indent=4)' \
  < broken.yaml > fixed.json
```

Llamadas al Sistema dup y dup2

Para ver cómo duplican las entradas en la Tabla de Canales (TC), vamos a probar los programas siguientes. En concreto, dup copia el canal indicado en la primera entrada libre de la TC, es decir, la que tiene el valor más bajo.

```
// // Comprobar que se modifica la Tabla de Canales (TC)
// gcc canales.c -o c      # compilamos
3 // ./c                  # arrancamos el programa
// ps                      # cuando paramos con CTRL-Z para mirar su TC
//                          # no hace falta más que la primera vez
6 // ls -l /proc/17165/fd # miramos la TC en este momento
// cat f1.txt

9 #include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
12 #include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
15 #include <string.h>

int main(){
18   char cad[100];
   printf("\n1. Canal 1\n"); // Salida estándar con lib de C

21   getchar(); // CTRL-Z para ver la TC, fg y pulsar tecla
```

```

24 // imprimir en un fichero con Llamadas al sistema
    int fd = creat("f1.txt", 0666);
    sprintf(cad, "\n2. Canal %d\n", fd);
    write(fd, cad, strlen(cad));
27 getchar(); // CTRL-Z para ver la TC, fg y pulsar tecla

    dup2(fd, 1);
30 // Ahora canales 1 y fd redireccionados al fichero
    getchar(); // CTRL-Z para ver la TC, fg y pulsar tecla

33 // a la vez printf en canal 1 y fichero
    printf("\n3. Canal 1\n"); // canal 1 y fichero f1.txt
    getchar(); // CTRL-Z para ver la TC, fg y pulsar tecla
36

// cerrar el fichero pero sigue el canal 1
    close(fd);
39 getchar(); // CTRL-Z para ver la TC, fg y pulsar tecla

    exit(0);
42 }

```

Fichero 4.7: canalesB.c

```

/* Prueba de distintos canales */
// gcc canalesC.c -o c2
3
#include <stdio.h>
#include <sys/types.h>
6 #include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
9 #include <stdlib.h>

int main(){
12 // The following example creates the file /tmp/file with read and
// write permissions for the file owner and read permission for group and
// others. The resulting file descriptor is assigned to the fd variable.
    close(1);
    int fd;
15 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    char *filename = "./fich";
    fd = creat(filename, mode);
18 printf("\n1. Canal %d\n", fd);
    fflush(stdout);
    close(fd);
21 dup(2);
    getchar();

24 fprintf(stdout, "\n2. Canal 1 y 2\n");
    exit(0);
}

```

Fichero 4.8: canalesC.c

4.5. Llamadas al Sistema e Inodos

Si hacemos `man 3 scandir` nos explica cómo mirar los ficheros y directorios de un directorio dado. Dentro tenemos este ejemplo:

```

3  /* imprimir ficheros en el directorio actual en orden inverso */
   #include <dirent.h>
   main() {
       struct dirent **namelist;
       int n;
6   n = scandir(".", &namelist, 0, alphasort);
       if (n < 0)
           perror("scandir");
9   else {
       while (n--) {
           printf("%s\n", namelist[n]->d_name);
12          free(namelist[n]);
       }
       free(namelist);
15  }
   }

```

Si llamamos al código `cdir.c`, compilaremos con `gcc -o cdir cdir.c` y ejecutaremos con `./cdir` pero necesitarás averiguar qué librerías hay que incluir para poder compilar. Usa `man` para encontrar esa información.

El programa crea una variable `namelist` en la que va a recibir como resultado de la llamada al sistema `scanlist` una tabla de punteros a estructuras `dirent`. Una *dirent* es una entrada de directorio, y podemos comprobar en internet que es un número de inodo y una cadena de caracteres.

Nuestro programa necesita obtener el tamaño en bytes de los ficheros que contiene un directorio, por lo que a partir del ejemplo anterior vamos a hacer un programa que lo haga. Encontramos en internet la página https://en.wikipedia.org/wiki/Stat_%28system_call%29 que nos sirve de base para conocer qué llamadas al sistema necesitamos.

También puedes hacer `man 2 stat` o ver <http://man7.org/linux/man-pages/man2/stat.2.html>

Mira esta *chuleta* de la línea de comandos.

Compila y ejecuta el programa anterior.

Escribe la modificación del programa anterior que hace la tarea descrita (imprimir los bytes de todos los ficheros) y pon comentarios en el código que expliquen dichas variaciones.