

Sistemas Operativos, Práctica 1: Memoria

Leandro García y Fabián Gutiérrez (Pareja 02)

21 de febrero de 2020

Ejercicio 1

a) Utilizando **man -k pthread** se obtiene:

```
pthread_cancel (3)    - send a cancellation request to a thread
pthread_cleanup_pop (3) - push and pop thread cancellation clean-up handlers
pthread_cleanup_pop_restore_np (3) - push and pop thread cancellation clean-u...
pthread_cleanup_push (3) - push and pop thread cancellation clean-up handlers
pthread_cleanup_push_defer_np (3) - push and pop thread cancellation clean-up...
pthread_create (3)    - create a new thread
pthread_detach (3)    - detach a thread
pthread_equal (3)     - compare thread IDs
pthread_exit (3)      - terminate calling thread
pthread_getaffinity_np (3) - set/get CPU affinity of a thread
pthread_getattr_default_np (3) - get or set default thread-creation attributes
pthread_getattr_np (3) - get attributes of created thread
pthread_getconcurrency (3) - set/get the concurrency level
pthread_getcpuclockid (3) - retrieve ID of a thread's CPU time clock
pthread_getname_np (3) - set/get the name of a thread
pthread_getschedparam (3) - set/get scheduling policy and parameters of a thread
pthread_join (3)      - join with a terminated thread
pthread_kill (3)      - send a signal to a thread
pthread_kill_other_threads_np (3) - terminate all other threads in process
pthread_mutex_consistent (3) - make a robust mutex consistent
pthread_mutex_consistent_np (3) - make a robust mutex consistent
pthread_mutexattr_getpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_getrobust (3) - get and set the robustness attribute of a m...
pthread_mutexattr_getrobust_np (3) - get and set the robustness attribute of ...
pthread_mutexattr_setpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_setrobust (3) - get and set the robustness attribute of a m...
pthread_mutexattr_setrobust_np (3) - get and set the robustness attribute of ...
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the t...
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the t...
pthread_self (3)      - obtain ID of the calling thread
pthread_setaffinity_np (3) - set/get CPU affinity of a thread
pthread_setattr_default_np (3) - get or set default thread-creation attributes
pthread_setcancelstate (3) - set cancelability state and type
pthread_setcanceltype (3) - set cancelability state and type
```

```

pthread_setconcurrency (3) - set/get the concurrency level
pthread_setname_np (3) - set/get the name of a thread
pthread_setschedparam (3) - set/get scheduling policy and parameters of a thread
pthread_setschedprio (3) - set scheduling priority of a thread
pthread_sigmask (3) - examine and change mask of blocked signals
pthread_sigqueue (3) - queue a signal and data to a thread
pthread_spin_destroy (3) - initialize or destroy a spin lock
pthread_spin_init (3) - initialize or destroy a spin lock
pthread_spin_lock (3) - lock and unlock a spin lock
pthread_spin_trylock (3) - lock and unlock a spin lock
pthread_spin_unlock (3) - lock and unlock a spin lock
pthread_testcancel (3) - request delivery of any pending cancellation request
pthread_timedjoin_np (3) - try to join with a terminated thread
pthread_tryjoin_np (3) - try to join with a terminated thread
pthread_yield (3) - yield the processor
pthreads (7) - POSIX threads

```

- b) Por otra parte, usando **man man** se obtiene la documentación del comando **man**, en particular que la sección 2 está dedicada a las llamadas al sistema. Luego con **man 2 write** se obtiene la documentación de **write**.

```

DESCRIPTION
man is the system's manual pager. Each page argument given to man is normally the name of a program, utility
or function. The manual page associated with each of these arguments is then found and displayed. A section,
if provided, will direct man to look only in that section of the manual. The default action is to search in
all of the available sections following a pre-defined order ("1 n l 8 3 2 3posix 3pm 3perl 3am 5 4 9 6 7" by
default, unless overridden by the SECTION directive in /etc/manpath.config), and to show only the first page
found, even if page exists in several sections.

The table below shows the section numbers of the manual followed by the types of pages they contain.

1 Executable programs or shell commands
2 System calls (functions provided by the kernel)
3 Library calls (functions within program libraries)
4 Special files (usually found in /dev)
5 File formats and conventions eg /etc/passwd
6 Games
7 Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8 System administration commands (usually only for root)
9 Kernel routines [Non standard]

WRITE(2) Linux Programmer's Manual WRITE(2)

NAME
write - write to a file descriptor

```

Figura 1: Salida de la ejecución de **man man** y **man 2 write**

Ejercicio 2

- a) Para buscar las líneas que contienen la palabra **molino** se utiliza el comando **grep**, cuya salida se redirige luego al fichero de destino deseado mediante el símbolo **>>**.
El comando resultante de esto es: **grep molino 'don quijote.txt' >> aventuras.txt**.
- b) Para obtener los ficheros contenidos en un directorio se utiliza el comando **ls**. Sobre la salida de este se aplica el comando **wc** con el modificador **-l** para contar el número de líneas, esto es, el número de elementos contenidos en el directorio.

Finalmente el comando utilizado es: **ls | wc -l**

- c) En primer lugar, se concatenan los ficheros con **cat**, redirigiendo la salida de error a **/dev/null** con **2>**. Luego, para obtener las líneas distintas (asumiendo que dos líneas iguales no-consecutivas no son distintas), se ordenan con **sort** y luego se ejecuta **uniq**, ya que este comando solo elimina líneas duplicadas consecutivas. Finalmente, se cuentan las líneas resultantes con **wc -l** y se redirige la salida a “num compra.txt” con **>**.

El comando queda: **cat 'lista de la compra Pepe.txt' 'lista de la compra Elena.txt' 2>/dev/null | sort | uniq | wc -l >'num compra.txt'**

- d) Primero se obtienen los hilos de todos los procesos del sistema utilizando el comando **ps** con los modificadores **-A -L**. Luego, se extrae de este resultado la primera columna, que es la correspondiente al PID asociado a cada hilo, con **awk**. Finalmente, con **uniq -c**, se cuentan las apariciones de cada PID, es decir, el número de hilos asociado a cada proceso, y se redirige al fichero “hilos.txt”.

El comando queda: **ps -A -L | awk '{print \$1}' | uniq -c >hilos.txt**

Antes de la llamada a **uniq** podría ser necesario hacer **sort** como en el apartado anterior, pero la salida de **ps** ya viene ordenada según el PID, por lo que se puede omitir este paso.

Ejercicio 3

- a) Una vez elaborado el programa en C, se ejecuta este pasándole como argumento un nombre de fichero inexistente. El mensaje de error generado en la salida de errores es “No such file or directory”, que se corresponde con el valor 2 en la variable **errno**.

```
leandro@LAPTOP-2LMQ7308:/mnt/c/Users/leand/Documents/GIT/soper/P1$ ./ej3.exe noexiste.txt
Error: No such file or directory
Valor de errno: 2
```

Figura 2: Resultado de la ejecución para un fichero inexistente

- b) Si se repite la ejecución introduciendo esta vez como parámetro el fichero **/etc/shadow** se puede observar que muestra el error **Permission denied**, equivalente al valor 13 de la variable **errno**.

```
leandro@LAPTOP-2LMQ7308:/mnt/c/Users/leand/Documents/GIT/soper/P1$ ./ej3.exe /etc/shadow
Error: Permission denied
Valor de errno: 13
```

Figura 3: Resultado de la ejecución para el fichero **/etc/shadow**

- c) Puesto que las funciones **fprintf** y **printf** no utilizan la variable **errno**, esta no será modificada al imprimir por pantalla, por lo que no sería necesario ningún cambio. Si se quisiera utilizar una función que utilizase la variable **errno**, se puede obtener inmediatamente después de usar **fopen** la descripción del error mediante la función **strerror** para usarla posteriormente.

Ejercicio 4

- a) Una vez ejecutado el proceso, este aparece reflejado en la terminal en la que se ejecuta el comando **top**. El proceso aparece en la primera posición de la lista ocupando casi un 100 % de CPU.

- b) Al ejecutar en terminal esta nueva versión del código el proceso aparece nuevamente como resultado del comando `top`. Sin embargo, en esta ocasión figura en el último lugar de la lista y con un consumo de recursos nulo (de hecho, en la columna `TIME` aparece que apenas consume tiempo, mientras que con `clock` consumía los 10 segundos y finalizaba).

Ejercicio 5

- a) Si el proceso no esperase a los hilos, podría finalizar antes que estos (o alguno de estos), en cuyo caso los hilos serían eliminados junto al proceso antes de haber terminado.
- b) Cambiar `exit` por `pthread_exit` hace que el hilo principal (que acaba llamándola) termine, mientras que los otros hilos pueden seguir ejecutándose (terminar el hilo principal no libera los recursos del proceso).
- c) Solo es correcto no esperar a hilos desligados, ya que se liberan automáticamente los recursos de estos en cuanto terminan (se entiende que no hay otros hilos que dependan de ellos), mientras que con el resto de hilos es necesario esperar al `join` correspondiente para liberar completamente sus recursos. Por tanto, habría que desligar los hilos que antes tenían que esperarse, por ejemplo, añadiendo en `slow_printf` el siguiente código:

```
int error = 0;

error = pthread_detach(pthread_self());
if(error != 0)
{
    fprintf(stderr, "pthread_detach: %s\n", strerror(error));
    pthread_exit(EXIT_FAILURE);
}
```

Ejercicio 6

Para este programa se definió la rutina `funcion` que realiza las acciones solicitadas (esperar un cierto tiempo, elevar un número al cubo, ...). Para poder pasar el tiempo de espera y el número de hilo a esta rutina en un único parámetro de entrada, se creó la estructura `thread_info` que almacena estos datos.

Ejercicio 7

- a) No, porque una vez se lanzan distintos procesos, se desconoce cómo va a gestionarlos el sistema operativo.
- b) Para conseguir mostrar el `PID` y el `PPID` de cada proceso hijo se declara una variable `ppid` de tipo `pid_t` en la cual, en cada iteración del bucle `for`, se guarda el `PID` del proceso en ejecución a través de la función `getpid`. Una vez se ha creado cada hijo se puede acceder desde él a esta variable, ya que el nuevo proceso será una copia exacta del proceso padre (en particular, se habrá hecho una copia de la variable en cuestión).

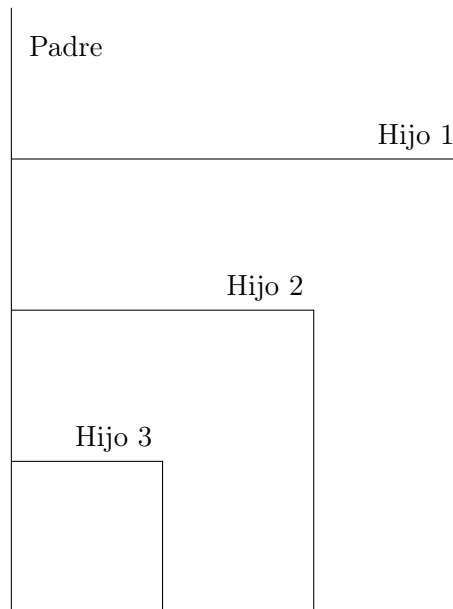
Para mostrarlo por pantalla se realiza también la siguiente modificación en el código.

```

else if(pid == 0)
{
    printf("Hijo_(PPID:_%d, _PID:_%d)\n", ppid, getpid());
    exit(EXIT_SUCCESS);
}

```

- c) El proceso contiene un bucle `for` con 3 iteraciones en el cual se llama a la función `fork`. En cada iteración, el proceso hijo resultante del `fork` imprime por pantalla un mensaje y finaliza, mientras que el padre pasa a la siguiente iteración. Con esto, el resultado es un único proceso padre del que parten 3 procesos hijos, tal como se muestra en el diagrama.



- d) Sí puede dejar huérfanos ya que `wait` espera un único cambio de estado de alguno de sus hijos, por lo que el resto de hijos podría quedar huérfano.
- e) Puesto que en el código es el proceso padre el único que vuelve a ejecutar el bucle y sigue produciendo procesos hijos, una posible solución sería poner al final del código un bucle con el mismo número de iteraciones para esperar a que finalicen todos los hijos.

```

for (i = 0; i < NUM_PROC; i++)
{
    wait(NULL);
}

```

Ejercicio 8

El programa desarrollado es similar al del apartado anterior, con la diferencia de que es el hijo quien vuelve a hacer `fork`, produciéndose el árbol de procesos esperado (en cascada).

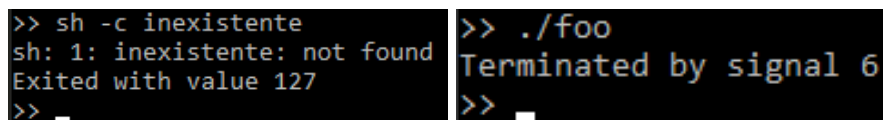
Ejercicio 9

- a) Al ejecutar el programa, se imprime en la salida estándar “Padre: ”. El programa no es correcto ya que, además de no liberar la cadena reservada, se copia el mensaje en el proceso hijo y se pretende que se imprima en el proceso padre. El error radica en que, al hacer `fork`, se crea una copia de los recursos del proceso padre, por lo que la variable `sentence` del hijo es distinta a la del padre y modificar una no altera la otra.
- b) Habría que añadir tanto en el proceso padre como en el proceso hijo sendas llamadas a `free` para evitar pérdidas de memoria, puesto que cada uno tiene una copia distinta de la cadena reservada inicialmente:

```
else if (pid == 0)
{
    strcpy(sentence, MESSAGE);
    free(sentence);
    exit(EXIT_SUCCESS);
}
else
{
    wait(NULL);
    printf("Padre: □ %s\n", sentence);
    free(sentence);
    exit(EXIT_SUCCESS);
}
```

Ejercicio 10

- b) Se utilizó la función `execvp` ya que permite buscar el nombre del fichero de la función que se pasa por argumento (p) y pasar los argumentos de esta mediante un vector (v), lo que concuerda con la funcionalidad proporcionada por `wordexp` (adapta una cadena de caracteres a un vector de argumentos, `we_wordv`, como hace la shell). Por tanto, podría usarse funciones de `exec` “sin p” si se indica directamente el path de la función a ejecutar y funciones que pasen los argumentos de esta por separado, como `printf` (l, en vez de v).
- c) Se imprime por pantalla “sh: 1: inexistente: not found”, “Exited with value 127”.
- d) Se imprime por pantalla “Terminated by signal 6”.



```
>> sh -c inexistente
sh: 1: inexistente: not found
Exited with value 127
>>

>> ./foo
Terminated by signal 6
>>
```

Figura 4: Salida para los apartados (c) y (d)

- e) El único cambio que hubo de hacerse fue reemplazar la llamada a `fork + execvp` por una llamada a `posix_spawn` (nótese el uso de funciones de ejecución “con p” en ambas versiones).

Ejercicio 11

- a) El nombre del ejecutable del proceso elegido es **top**. Esta información se encuentra en el enlace **exe**, el cual se puede analizar con la instrucción **readlink**.
- b) De la misma forma que en el apartado a) se puede ver que el directorio del proceso es **/usr/bin/top**.
- c) La línea de comandos utilizada es **top -d 10**, que se encuentra en el fichero **cmdline**.
- d) El valor de las variables de entorno se encuentra en el fichero **environ**, donde se encuentra **LANG=es_ES.UTF-8**.
- e) Para hallar la lista de hilos del proceso se puede acceder al directorio **task**, donde se encuentran enlaces a todos sus hilos. En este caso se encuentra dentro del directorio el PID 1809, es decir, sólomente se encuentra el hilo principal.

Ejercicio 12

- a) Dentro del directorio **fd** se encuentran solamente los descriptors de fichero 0, 1 y 2, correspondientes a los ficheros **stdin**, **stdout** y **stderr**.
- b) Una vez se llega al *Stop 2* se añade a la lista de descriptors de fichero el número 3. En el *Stop 3* se añade el descriptor de fichero 4.
- c) La función **unlink** eliminó la ruta de acceso (desde el sistema de archivos) al fichero **FILE1**, mas no se eliminó el fichero ya que el proceso en ejecución lo tiene abierto. De hecho, el fichero aparece en el directorio de **/proc** del proceso (aunque indica que está eliminado), y desaparecerá cuando este lo cierre al ser el último proceso que tiene abierto el fichero. Si se quiere recuperar, podría hacerse un **cat** o un **less** y redirigir la salida a un fichero nuevo: **cat 3 >[ruta del nuevo fichero]** (el fichero tiene descriptor 3).
- d) Tras la quinta parada del programa, al realizarse un **close** sobre **FILE1**, descriptor de fichero 3 desaparece del proceso. Tras abrir dos nuevos ficheros, en los *Stops* 6 y 7, aparecen los descriptors de fichero 3 (nuevamente) y 5, por lo que se puede deducir que una vez un descriptor es eliminado su número puede ser reutilizado por otro fichero.

Ejercicio 13

- a) El mensaje *Yo soy tu padre* se almacena en el *buffer* del fichero, pero no llega a imprimirse por pantalla antes del **fork**. Posteriormente, el proceso hijo añade al *buffer* el mensaje *Noooooo* y lo imprime por pantalla al finalizar (*Yo soy tu padreNoooooo*); mientras que el padre, una vez acabe, mostrará el primer mensaje de nuevo por pantalla. Con esto, el mensaje aparecerá por pantalla dos veces.
- b) El mensaje *Yo soy tu padre* se muestra por pantalla en una sola ocasión, ya que se encuentra antes de la llamada del **fork** y, al haber introducido saltos de línea, este se imprime de forma inmediata.
- c) Al redirigir la salida a un fichero se presenta nuevamente el problema del apartado a), puesto que el salto de línea es interpretado como un carácter y se almacena igualmente en un *buffer* antes del **fork**.

- d) El problema se puede solucionar si se obliga al programa a volcar el contenido del *buffer* inmediatamente después del `printf` utilizando la función `fflush`.

Ejercicio 14

- a) Se muestran correctamente los mensajes del padre y del hijo, aunque se desconoce en que orden aparecerán.
- b) El proceso no termina, ya que el **pipe** aún tiene abiertos sus extremos de lectura y escritura, a los que tiene acceso el proceso padre, por lo que el extremo lector queda a la espera de que el extremo escritor añada algo a la tubería (y no se da el caso de que el proceso acabe con una señal **SIGPIPE** producto de escribir en un **pipe** sin lectores).
- c) No se imprime nada, ya que el padre no se ve forzado a esperar a que haya algo en la tubería que leer y tampoco hace `wait`, y por tanto logra finalizar dentro del segundo que ha de esperar el hijo antes de escribir. Nótese que el hijo, al terminar su padre antes, se queda sin imprimir y huérfano.

Ejercicio 15

En este ejercicio se elabora un programa en C en el cual se crean dos hijos y dos pipes que comunican al proceso padre con cada uno de ellos. En el primero hijo se genera un entero aleatorio que será pasado al padre a través de un pipe. Desde el padre, se leera el entero del primer pipe y se pasará al segundo hijo por el segundo pipe. Por último, en el segundo hijo, se imprimirá el entero en un fichero llamado "`numero_leido.txt`". Se ha de tener en cuenta también que el proceso no termine dejando pipes o ficheros abiertos ni hijos huérfanos.