

# Memoria Sistemas Operativos

Miguel Ibáñez y Sergio Hidalgo, Grupo 2273, Pareja 05

Ejercicio 1: Uso del manual.

a) Buscar en el manual la lista de funciones disponibles para el manejo de hilos y copiarla en la memoria junto con el comando usado para mostrarla. Las funciones de manejo de hilos comienzan por pthread.

**man -k pthread**

```
sergio@seregio-pc:~/Escritorio/unl/2o/2o_cuatr/SOPER-P/practicas/p1$ man -k pthread
pthread_attr_destroy (3) - initialize and destroy thread attributes object
pthread_attr_getaffinity_np (3) - set/get CPU affinity attribute in thread attributes object
pthread_attr_getdetachstate (3) - set/get detach state attribute in thread attributes object
pthread_attr_getguardsize (3) - set/get guard size attribute in thread attributes object
pthread_attr_getinheritsched (3) - set/get inherit-scheduler attribute in thread attributes object
pthread_attr_getschedparam (3) - set/get scheduling parameter attributes in thread attributes object
pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in thread attributes object
pthread_attr_getscope (3) - set/get contention scope attribute in thread attributes object
pthread_attr_getstack (3) - set/get stack attributes in thread attributes object
pthread_attr_getstackaddr (3) - set/get stack address attribute in thread attributes object
pthread_attr_getstacksize (3) - set/get stack size attribute in thread attributes object
pthread_attr_init (3) - initialize and destroy thread attributes object
pthread_attr_setaffinity_np (3) - set/get CPU affinity attribute in thread attributes object
pthread_attr_setdetachstate (3) - set/get detach state attribute in thread attributes object
pthread_attr_setguardsize (3) - set/get guard size attribute in thread attributes object
pthread_attr_setinheritsched (3) - set/get inherit-scheduler attribute in thread attributes object
pthread_attr_setschedparam (3) - set/get scheduling parameter attributes in thread attributes object
pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in thread attributes object
pthread_attr_setscope (3) - set/get contention scope attribute in thread attributes object
pthread_attr_setstack (3) - set/get stack attributes in thread attributes object
pthread_attr_setstackaddr (3) - set/get stack address attribute in thread attributes object
pthread_attr_setstacksize (3) - set/get stack size attribute in thread attributes object
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-up handlers while saving cancelability type
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 handlers while saving cancelability type
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 mutex attributes object
pthread_mutexattr_getrobust_np (3) - get and set the robustness attribute of a mutex attributes object
pthread_mutexattr_setpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_setrobust (3) - get and set the robustness attribute of a mutex attributes object
pthread_mutexattr_setrobust_np (3) - get and set the robustness attribute of a mutex attributes object
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the thread read-write lock attribute object
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the thread read-write lock attribute object
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) Consultar en la ayuda en qué sección del manual se encuentran las llamadas al sistema y buscar información sobre la llamada al sistema write. Escribir en la memoria los comandos usados.

**man man**

```
DESCRIPCIÓN
man es el paginador de manuales del sistema. Cada argumento de página argumento dado a man normalmente es e
estos argumentos es, pues, encontrada y mostrada. Si se proporciona una sección, man mirará solo en esa sec
siguiendo un orden predefinido (véase DEFAULTS), y mostrar solo la primera página encontrada, incluso si la

La tabla de abajo muestra los números de sección del manual seguidos por los tipos de página que contienen.

1 Programas ejecutables u órdenes de la shell
2 Llamadas al sistema (funciones proporcionados por el núcleo)
3 Llamadas a biblioteca (funciones dentro de bibliotecas de programa)
4 Ficheros especiales (normalmente se encuentran en /dev)
5 Formatos de fichero y convenios, p.e. /etc/passwd
6 Juegos
7 Miscelánea (incluidos paquetes de macros y convenios), p.e. man(7), groff(7)
8 Órdenes de administración del sistema (normalmente solo para root)
9 Rutinas del núcleo [No estándar]
```

**man 2 write**

```
WRITE(2) Linux Programmer's Manual WRITE(2)
NAME
write - write to a file descriptor
SYNOPSIS
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
The number of bytes written may be less than count if, for example, there is insufficient space on the underlying physical medium, or the RLIMIT_PSIZE resource limit is encountered (see setrlimit(2)), or the call was interrupted by a signal handler after having written less than count bytes. (See also pipe(7).)
For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was opened with O_APPEND, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.
POSIX requires that a read(2) that can be proved to occur after a write() has returned will return the new data. Note that not all filesystems are POSIX conforming.
According to POSIX.1, if count is greater than SSIZE_MAX, the result is implementation-defined; see NOTES for the upper limit on Linux.
RETURN VALUE
On success, the number of bytes written is returned. On error, -1 is returned, and errno is set to indicate the cause of the error.
Note that a successful write() may transfer fewer than count bytes. Such partial writes can occur for various reasons; for example, because there was insufficient space on the disk device to write all of the requested bytes, or because a blocked write() to a socket, pipe, or similar was interrupted by a signal handler after it had transferred some, but before it had transferred all of the requested bytes. In the event of a partial write, the caller can make another write() call to transfer the remaining bytes. The subsequent call will either transfer further bytes or may result in an error (e.g., if the disk is now full).
If count is zero and fd refers to a regular file, then write() may return a failure status if one of the errors below is detected. If no errors are detected, or error detection is not performed, 0 will be returned without causing any other effect. If count is zero and fd refers to a file other than a regular file, the results are not specified.
```

Ejercicio 2: Comandos y redireccionamiento.

a) Escribir un comando que busque las líneas que contengan molino en el fichero don quijote.txt y las añade al final del fichero aventuras.txt. Copiar el comando en la memoria, justificando las opciones utilizadas.

**grep -w molino don\ quijote.txt >> aventuras.txt**

grep -w : Selecciona las líneas que contengan únicamente la palabra exacta

>> : se usa para añadir la salida a otro fichero

b) Elaborar un pipeline que cuente el número de ficheros en el directorio actual. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.

**ls | wc -l**

ls : sirve para listar

wc -l: cuenta el número de líneas de la terminal (en este caso el número de salidas del comando ls)

c) Elaborar un pipeline que cuente el número de líneas distintas al concatenar lista de la compra Pepe.txt y lista de la compra Elena.txt y lo escriba en num compra.txt. Si alguno de los ficheros no existe, hay que ignorar los mensajes de error, para lo cual se redirigirá la salida de errores a /dev/null. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.

**cat "lista de la compra Pepe.txt" "lista de la compra Elena.txt" 2> /dev/null | sort | uniq | wc -l > "num compra.txt"**

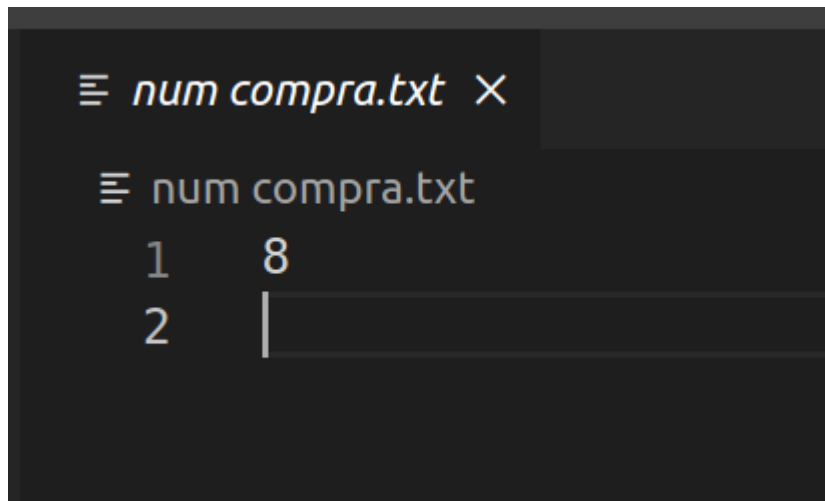
cat: concatena ficheros

2> : establece la salida de los dos ficheros a errores en null

sort: ordena

uniq: filtra las coincidencias

wc -l > "nombre fichero" : printea el número de líneas de la salida en el fichero.



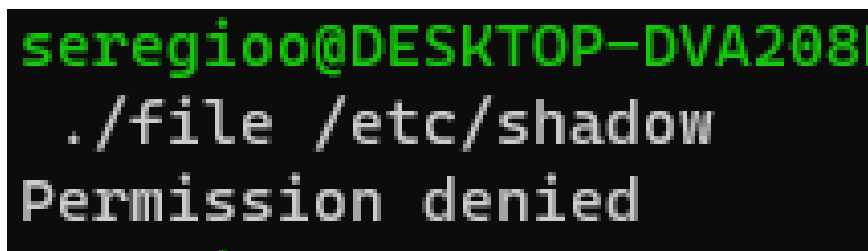
```
num compra.txt X
num compra.txt
1      8
2      |
```

Ejercicio 3: Control de errores. Escribir un programa que abra un fichero indicado por el primer parámetro en modo lectura usando la función fopen. En caso de error de apertura, el programa mostrará el mensaje de error correspondiente por pantalla usando perror.

a) ¿Qué mensaje se imprime al intentar abrir un fichero inexistente? ¿A qué valor de errno corresponde?

No such file or directory, al ENOENT 2.

b) ¿Qué mensaje se imprime al intentar abrir el fichero /etc/shadow? ¿A qué valor de errno corresponde?



```
seregioo@DESKTOP-DVA2081
./file /etc/shadow
Permission denied
```

Al error EACCES 13

c) Si se desea imprimir el valor de errno antes de la llamada a perror, ¿qué modificaciones se deberían realizar para garantizar que el mensaje de perror se corresponde con el error de fopen?

"perror("fopen");" de esta manera aparecerá la cadena fopen y seguido el error.

#### Ejercicio 4: Espera activa e inactiva.

a) Escribir un programa que realice una espera de 10 segundos usando la función clock en un bucle. Ejecutar en otra terminal el comando top. ¿Qué se observa?

Que se le da el 100 % de la CPU en ciertos momentos

```
int main()
{
    time_t t = 0, t2 = 0;
    int i = 0;

    t = clock();
    for (t = 0; t < 10; i++)
    {
        t2 = clock();

        if (10 == ((double)(t2 - t) / CLOCKS_PER_SEC))
        {
            exit(EXIT_SUCCESS);
        }
    }

    exit(EXIT_SUCCESS);
}
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
11063	seregioo	20	0	2356	588	524	R	100.0	0.0	0:03.47	clock
1	root	20	0	896	532	464	S	0.0	0.0	0:00.00	init

b) Reescribir el programa usando sleep y volver a ejecutar top. ¿Ha cambiado algo?  
Si, ahora el programa no usa CPU, porque se ejecuta, y la función sleep lo deja inactivo durante 10 segs, después de eso, el programa finaliza.

```
int main()
{

    sleep(10);
    exit(EXIT_SUCCESS);
}
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
249	seregioo	20	0	584336	41300	28024	S	0.0	0.2	0:00.99	node
256	root	20	0	904	88	20	S	0.0	0.0	0:00.00	init
257	root	20	0	904	96	20	S	0.0	0.0	0:01.46	init
258	seregioo	20	0	593096	48144	28160	S	0.0	0.2	0:03.37	node
265	seregioo	20	0	828340	41176	29640	S	0.0	0.2	0:00.86	node
272	seregioo	20	0	989008	140720	34176	S	0.0	0.5	0:35.52	node
425	seregioo	20	0	2179764	67736	15736	S	0.0	0.3	0:16.46	cpptools
829	seregioo	20	0	581496	38484	29548	S	0.0	0.1	0:00.29	node
10745	seregioo	20	0	10872	3720	3208	R	0.0	0.0	0:00.26	top
10746	root	20	0	896	88	20	S	0.0	0.0	0:00.00	init
10747	root	20	0	896	88	20	S	0.0	0.0	0:00.01	init
10748	seregioo	20	0	10292	5440	3528	S	0.0	0.0	0:00.06	bash
10810	seregioo	20	0	5545848	29888	13200	S	0.0	0.1	0:01.02	cpptools-srv
10847	seregioo	20	0	5545980	19216	10972	S	0.0	0.1	0:00.06	cpptools-srv
11248	seregioo	20	0	5545848	26500	9892	S	0.0	0.1	0:00.05	cpptools-srv
11279	seregioo	20	0	5545848	27500	10144	S	0.0	0.1	0:00.06	cpptools-srv
11338	seregioo	20	0	2356	592	524	S	0.0	0.0	0:00.00	<u>clock</u>

Ejercicio 5: Finalización de hilos.

a) ¿Qué hubiera pasado si el proceso no hubiera esperado a los hilos? Para probarlo basta eliminar las llamadas a `pthread_join`.

El programa finaliza nada más empezar.

```
sergio@seregio-pc:~/Escritorio/unl/2o/2o cuatri/SOPER-P/practicas/p1$ ./thread_example
Program ./thread_example finished correctly
```

b) Con el código modificado del apartado anterior, indicar qué ocurre si se reemplaza la función `exit` por una llamada a `pthread_exit`.

Primero se acaba el programa y luego se espera a los hilos, lo que ocurre es que se mezclan las letras de forma pseudoaleatoria ya que el procesador decide según el reloj cuando le “toca” a qué hilo.

```
sergio@seregio-pc:~/Escritorio/uni/2o/2o cuatri/SOPER-P/practicas/p1$ ./thread_e
example
Program ./thread_example finished correctly
H W o e r l l d o sergio@seregio-pc:~/Escritorio/uni/2o/2o cuatri/SOPER-P/pr
```

c) Tras eliminar las llamadas a `pthread_join` en los apartados anteriores, el programa es ahora incorrecto porque no se espera a que terminen todos los hilos. Escribir en la memoria el código que será necesario añadir para que sea correcto no esperar a los hilos creados.

Hay que añadir un `pthread_join` para el primer hilo (Hello) antes de crear el segundo (World), para que lo imprima y no sucedan ambos hilos a la vez.

```
sergio@seregio-pc:~/Escritorio/uni/2o/2o cuatri/SOPER-P/practicas/p1$ ./thread_e
example
H e l l o   W o r l d Program ./thread_example finished correctly
```

## Ejercicio 6: Creación de procesos

a) Analizar el texto que imprime el programa. ¿Se puede saber a priori en qué orden se imprimirá el texto? ¿Por qué?

Si, primero imprimirá si hay un error, después "child" y la interacción del bucle y después "Parent" y el mismo valor, este bucle se repetirá según el número de procesos (cuyo valor es 3, pues así está definido en la macro)

```
sergio@seregio-pc:~/Escritorio/uni/2o/2o cuatri/SOPER-P/practicas/p1$ ./proc_e
mple wo-l
Parent 0
Child 0
Parent 1
Child 1
Parent 2
Child 2
```

b) Cambiar el código para que el hijo imprima su PID y el de su padre en vez de la variable i. Copiar las modificaciones en la memoria y explicarlas.

Se obtiene dentro del hijo el PID (el id del proceso hijo) y el PPID (el id del proceso padre) y se imprimen. En el padre se imprime la interacción del bucle.



```

else if (pid == 0) {

    pid = getpid();
    ppid = getppid();

    printf("Child PID=%d, Parent PID=%d\n", pid, ppid );
    exit(EXIT_SUCCESS);
}
else if (pid > 0) {
    printf("Parent %d\n", i);
}

```

```

sergio@seregio-pc:~/Escritorio/unt/2o/2o cuatr1/SOPER-P/practicas/pi$ ./proc_exa
mple
Parent 0
Child PID=13137, Parent PID=13136
Parent 1
Child PID=13138, Parent PID=13136
Parent 2
Child PID=13139, Parent PID=13136

```

c) Dados los dos siguientes diagramas de árbol:

¿A cuál de los dos árboles de procesos se corresponde el código de arriba, y por qué?

Es el de la izquierda dado que es un bucle y por lo tanto todos los procesos hijos van a derivar del proceso padre (el bucle).

¿Qué modificaciones habría que hacer en el código para obtener el otro árbol de procesos?

Añadiendo "pid = fork()" dentro de la condición if(pid == 0).

d) El código original deja procesos huérfanos, ¿por qué?

Para que deje procesos huérfanos se ha tenido que aumentar al número de procesos y llamar a la función sleep con 2 segundos antes de obtener los PID's , el motivo es que se queda esperando el hijo dentro de la condición pero el bucle sigue avanzando (proceso padre), de manera que el bucle finaliza y todavía hay procesos hijos dentro de la condición (el proceso padre termina y los hijos no han terminado).



```

sergio@sergio-pc: ~/Escritorio/uni/2o/2o cuatri/SOPER-P/practicas/pi$ ./proc_exa
mple
Parent 0: makefile
Parent 1:
Parent 2: num compra.txt
Parent 3:
Parent 4: pipe_example.c
Parent 5:
Parent 6: pow.c
Parent 7:
Parent 8: pow.h
Parent 9:
Child PID=13428, Parent PID=13427
Child PID=13429, Parent PID=13427
Child PID=13430, Parent PID=13427
Child PID=13431, Parent PID=13427
Child PID=13432, Parent PID=13427
Child PID=13433, Parent PID=13427
Child PID=13434, Parent PID=13427
Child PID=13435, Parent PID=1408
Child PID=13436, Parent PID=1408
Child PID=13437, Parent PID=1408

```

e) Introducir el mínimo número de cambios en el código para que no deje procesos huérfanos. Copiar las modificaciones en la memoria y explicarlas.

Poniendo un wait al principio del bucle se evitan los procesos huérfanos.

```

sergio@DESKTOP-DVA208M: /mnt/c/Users/sergi/Desktop/cositas/uni/2o/2o cuatri/SOPER-P/Practicas-Soper/pi$ ./proc_example
Parent 0
Child PID=870, Parent PID=869
Parent 1
Child PID=871, Parent PID=869
Parent 2
Child PID=872, Parent PID=869
Parent 3
Child PID=873, Parent PID=869
Parent 4
Parent 5
Child PID=874, Parent PID=869
Parent 6
Child PID=875, Parent PID=869
Child PID=876, Parent PID=869
Parent 7
Child PID=877, Parent PID=869
Parent 8
Child PID=878, Parent PID=869
Parent 9

```

Ejercicio 7: Espacio de memoria.

a) En el programa anterior se reserva memoria en el proceso padre y se inicializa en el proceso hijo usando la función strcpy (que copia un string a una posición de memoria). Una vez el proceso hijo termina, el padre lo imprime por pantalla. ¿Qué ocurre cuando se ejecuta el código? ¿Es este programa correcto? ¿Por qué?

Al ejecutarse la frase no llega a salir, porque fork () genera un proceso hijo, que retornará 0 (copiando en la cadena el valor definido en la macro) y mantendrá al proceso padre (cuyo PID será el del hijo, por lo que la cadena no tendrá ningún valor, pero hará el printf).

```

sergio@DESKTOP-DVA208M: /mnt/c/Users/sergi/Desktop/cositas/uni
Parent:

```

b) El programa anterior contiene una fuga de memoria ya que el array `sentence` nunca se libera. Corregir el código para eliminar esta fuga y copiar las modificaciones en la memoria. ¿Donde hay que liberar la memoria, en el proceso padre, en el hijo o en ambos? ¿Por qué?

```
/mrush <target_init> <Rounios> <n_threads>
```

El proceso hijo creado por `fork()`, es idéntico al padre pero no comparten memoria, por tanto cada proceso habrá reservado memoria distinta para cada cadena y se deberá liberar la memoria en ambos (evidentemente, en caso de error también se deberá liberar).

```
int main(void) {
    pid_t pid;
    char * sentence = calloc(sizeof(MESSAGE), 1);

    pid = fork();

    if (pid < 0) {
        perror("fork");
        free(sentence);
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        strcpy(sentence, MESSAGE);
        free(sentence);
        exit(EXIT_SUCCESS);
    }
    else {
        printf("Parent: %s\n", sentence);
        wait(NULL);
        free(sentence);
        exit(EXIT_SUCCESS);
    }
}
```

```
sergio@DESKTOP-DVA208M:/mnt/c/Users/sergi/Desktop/cositas/uni/2o/2o cuatri/SOPER-P/Practicas-Soper/pi$ valgrind --leak
-check=full ./proc_malloc
==9496== Memcheck, a memory error detector
==9496== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9496== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9496== Command: ./proc_malloc
==9496==
Parent:
==9497==
==9497== HEAP SUMMARY:
==9497==    in use at exit: 0 bytes in 0 blocks
==9497==    total heap usage: 1 allocs, 1 frees, 6 bytes allocated
==9497==
==9497== All heap blocks were freed -- no leaks are possible
==9497==
==9497== For lists of detected and suppressed errors, rerun with: -s
==9497== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==9496==
==9496== HEAP SUMMARY:
==9496==    in use at exit: 0 bytes in 0 blocks
==9496==    total heap usage: 2 allocs, 2 frees, 1,030 bytes allocated
==9496==
==9496== All heap blocks were freed -- no leaks are possible
==9496==
==9496== For lists of detected and suppressed errors, rerun with: -s
==9496== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio 8: Ejecución de programas. Dado el siguiente código en C:

a) ¿Qué sucede si se sustituye el primer elemento del array argv por la cadena "mi-ls"?  
¿Por qué?

No cambia la ejecución del programa porque se hace un fork y en el caso de que devuelva 0 (es decir, de que sea el proceso hijo) se ejecutará la función execvp que ejecutará el comando que se le pase por parámetro seguido de el contenido de la segunda posición de la array, el primero es el nombre que se le da al comando (pero no el que se ejecuta) y el tercero es parámetro NULL.

```
int main(void) {  
    char *argv[3] = {"mi-ls", "./", NULL};  
    pid_t pid;
```

```
sergio@seregio-pc: ~/Escritorio/unl/2o/2o cuatr1/SOPER-P/practicas/p1$ ./proc_exec  
aventuras.txt      file_descriptors.c  pipe_example.c     proc_exec.c  
clock.c            'lista de la compra Elena.txt'  pow.c              proc_exec.o  
'don quijote.txt'  'lista de la compra Pepe.txt'  pow.h              proc_malloc.c  
file_buffer.c      makefile            proc_example.c     thread_example.c  
file.c             'num compra.txt'     proc_exec
```

b) Indicar las modificaciones que habría que hacer en el programa anterior para utilizar la función exec en lugar de execvp.

Hay que cambiar "ls" por la ruta de ls (que se encuentra mediante "which ls") y los argumentos deben ir separados, además se cambia el nombre de perror por si falla, que muestre el comando que se está utilizando ahora.

```
else if (pid == 0) {  
    if (execl("/usr/bin/ls", argv[0], argv[1], argv[2])) {  
        perror("execl");  
        exit(EXIT_FAILURE);  
    }  
}
```

No cambia la ejecución del programa porque se hace un fork y en el caso de  
Ejercicio 9: Directorio de información de procesos. Buscar para alguno de los procesos la siguiente información en el directorio /proc y escribir tanto la información como el fichero utilizado en la memoria. Hay que tener en cuenta que tanto las variables de entorno como la lista de comandos delimitan los elementos con \0, así que puede ser conveniente convertir los \0 a \n usando tr "\0"\n".

Comando top:

a) El nombre del ejecutable  
top

```
seregioo@DESKTOP-DVA208M:/usr$ which top  
/usr/bin/top
```

b) El directorio actual del proceso

```
489 seregioo 20 0 10872 3656 3144 R 0.0 0.0 0:00.10 top
/proc/489
```

c) La línea de comandos que se usó para lanzarlo

top

d) El valor de la variable de entorno LANG

LANG = C.UTF-8

```
seregioo@DESKTOP-DVA208M:/etc/profile.d$ locale -a
C
C.UTF-8
POSIX
en_US.utf8
```

```
seregioo@DESKTOP-DVA208M:/etc/profile.d$ locale
LANG=C.UTF-8
LANGUAGE=
LC_CTYPE="C.UTF-8"
LC_NUMERIC="C.UTF-8"
LC_TIME="C.UTF-8"
LC_COLLATE="C.UTF-8"
LC_MONETARY="C.UTF-8"
LC_MESSAGES="C.UTF-8"
LC_PAPER="C.UTF-8"
LC_NAME="C.UTF-8"
LC_ADDRESS="C.UTF-8"
LC_TELEPHONE="C.UTF-8"
LC_MEASUREMENT="C.UTF-8"
LC_IDENTIFICATION="C.UTF-8"
LC_ALL=
```

e) La lista de hilos del proceso

Sólo hay 1 hilo.

```
seregioo@DESKTOP-DVA208M:/proc/489$ cat status | grep Threads
Threads: 1
```

```
GNU nano 4.8                                status
Name:    top
Umask:   0022
State:   S (sleeping)
Tgid:    489
Ngid:    0
Pid:     489
PPid:    476
TracerPid: 0
Uid:     1000    1000    1000    1000
Gid:     1000    1000    1000    1000
FDSize:  256
Groups:  4 20 24 25 27 29 30 44 46 117 1000
NStgid:  489
NSpid:   489
NSpgid:  489
NSsid:   476
VmPeak:  10908 kB
VmSize:  10872 kB
VmLck:   0 kB
VmPin:   0 kB
VmHWM:   3656 kB
VmRSS:   3656 kB
RssAnon:           512 kB
RssFile:          3144 kB
RssShmem:          0 kB
VmData:   948 kB
VmStk:    132 kB
VmExe:     92 kB
VmLib:   3440 kB
VmPTE:     60 kB
VmSwap:    0 kB
HugetlbPages:      0 kB
CoreDumping: 0
THP_enabled: 1
Threads:    1
SigQ:  0/102454
[ File 'status' is unwritable ]
```

Ejercicio 10: Visualización de descriptores de fichero.

a) Stop 1. Inspeccionar los descriptores de fichero del proceso. ¿Qué descriptores de fichero se encuentran abiertos?

stdin (0), stdout (1) y stderr(2), que son respectivamente la entrada estándar, la salida estándar y la salida de errores.

```
seregioo@DESKTOP-DVA208M:/mnt/c/Users/sergi/Desktop/cositas/uni/2o/2o cuatri/SOPER-P/Practica
ors
PID = 833
Stop 1
```

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2
```

¿A qué tipo de fichero apuntan?

Son tres descriptores de fichero que se abren al iniciarse el programa, por tanto apuntan al programa ejecutándose (fichero de lectura escritura).

b) Stop 2 y Stop 3. ¿Qué cambios se han producido en la tabla de descriptores de fichero?

En el Stop 2 se crea un nuevo descriptor de ficheros en la tabla, porque se crea un fichero en el programa.

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2 3
```

Y en el Stop 3 otro fichero más

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2 3 4
```

c) Stop 4. ¿Se ha borrado del disco el fichero FILE1? ¿Por qué? ¿Se sigue pudiendo acceder al fichero a través del directorio /proc? ¿Hay, por tanto, alguna forma sencilla de recuperar los datos?

No, en el Stop 4 se borra una ruta para acceder al fichero mediante el nombre con “unlink”, pero todavía es accesible desde /proc hasta que no se rompan todas las rutas al mismo. Se puede recuperar, mediante la función link, pero se debe hacer antes de que el programa cierre el fichero, de lo contrario quedará inaccesible.

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2 3 4
```

d) Stop 5, Stop 6 y Stop 7. ¿Qué cambios se han producido en la tabla de descriptores de fichero?

Stop 5, se cierra el fichero al que se le había hecho “unlink” y queda inaccesible

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2 4
```

Stop 6 se abre otro fichero

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2 3 4
```

Stop 7 se abre un último fichero

```
seregioo@DESKTOP-DVA208M:/proc/833/fd$ ls
0 1 2 3 4 5
```

¿Que se puede deducir sobre la numeración de un descriptor de fichero obtenido tras una llamada a open?

Que empieza en 3 y que si hay un número sin usar dentro de la tabla, se utilizará para ese fichero (independientemente de que sea menor que el número de un fichero abierto anteriormente), es decir, los números de la tabla no representan el orden de apertura del fichero.

Ejercicio 11: Problemas con el buffer. Dado el siguiente código en C:

a) ¿Cuántas veces se escribe el mensaje <<I am your father>> por pantalla? ¿Por qué?  
2 veces, porque antes de hacer hacer “fork ()”, se guarda <<I am your father>>, después el proceso hijo ejecuta el programa y devuelve 0, por lo que se guarda en el buffer <<NOOOOO>>, y al acabar el programa se imprime todo por pantalla (de manera que la primera frase aparece 2 veces).

```
sergio@seregio-pc:~/Escritorio/un1/2o/2ocuatri/SOPER-P/practicas/p1$ ./file_buff
I am your fatherNooooooI am your father
```

b) En el programa falta el terminador de línea (\n) en los mensajes. Corregir este problema. ¿Sigue ocurriendo lo mismo? ¿Por qué?

```
sergio@seregio-pc:~/Escritorio/un1/2o/2ocuatri/SOPER-P/practicas/p1$ ./file_buff
er
I am your father
Noooooo
```

Ocorre lo mostrado en la imagen, pues al acabar con \n obliga a vaciar el buffer, por lo que solo se imprime una vez la primera frase.

c) Ejecutar el programa redirigiendo la salida a un fichero. ¿Qué ocurre ahora? ¿Por qué?



```
file_buffer.txt U X
file_buffer.txt
1 I am your father
2 Noooooo
3 I am your father
4
./file_buffer > file_buffer.txt
```

Vuelve a aparecer la anomalía porque al redirigir la salida a un fichero, el `\n` no vacía el buffer.

d) Indicar en la memoria como se puede corregir definitivamente este problema sin dejar de usar `printf`.

Haciendo un `fflush()` para vaciar el buffer antes de la bifurcación el problema se soluciona.

```
int main(void) {
    pid_t pid;

    printf("I am your father\n");
    fflush(stdout);

    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Noooooo\n");
        exit(EXIT_SUCCESS);
    }

    wait(NULL);
    exit(EXIT_SUCCESS);
}
```

```
file_buffer.txt
1 I am your father
2 Noooooo
3 |
```

## Ejercicio 12: Ejemplo de tuberías

a) Ejecutar el código. ¿Que se imprime por pantalla?

```
sergio@seregio-pc: ~/Escritorio/uni/2o/2ocuatri/SOPER-P/practicas/p1$ ./pipe_example
I have received the string: Hi, all! father
I have written in the pipe
```

b) ¿Qué ocurre si el proceso padre no cierra el extremo de escritura? ¿Por qué?.

```
    else {  
        /* Close of the write end in the parent. */  
        /*close(fd[1]); */  
        /* The message is read. */  
        nbytes = 0;  
        do {  
            nbytes = read(fd[0], readbuffer, sizeof(readbuffer));  
            if (nbytes == 0) break;  
        } while (nbytes > 0);  
    }  
}
```

sergio@seregio-pc:~/Escritorio/uni/2o/2ocuatru/SOPER-P/practicas/p1\$ ./pipe\_examp  
I have received the string: Hi all!  
I have written in the pipe

el programa sigue ejecutándose porque al no cerrar el extremo de escritura, no llega la señal EOF y por tanto, no se detecta el final de la cadena.mj