Práctica 1: Shell, Procesos, ficheros y tuberías

Autores: Marcos Aarón Bernuy, Kevin de la Coba Malam. Pareja 04 del grupo 2292.

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".*

```
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_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 threadspthread_testcancel (3) - request delivery of
any pending cancellation request
```

Comando: `man -k pthread > pthread.txt`

Con el commando man -k pthread buscamos toda la información en relación a pthread, al añadir el '>' redirigimos la salida de esta información al archivo "pthread.txt".

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.*
Si ejecutamos `man man` podemos ver que las llamadas al sistema están en la sección 2 del manual. Si usamos `man 2 write`, obtenemos toda la información con respecto las llamadas al sistema de write.

Ejercicio 2: Comandos y redireccionamiento.

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

`Grep molino don\ \quijote.txt >> aventuras.txt`
Grep se encarga de buscar la palabra molino que exista dentro del archivo aventuras.txt, luego con ">>" redirigimos la salida al final del archivo de aventuras.txt, básicamente hacemos un "apend".

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` nos devuelve los archivos, y `wc -l` cuenta las líneas.

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 redirigiría la salida de errores a /dev/null. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.*

```
cat lista\ de\ la\ compra\ Elena.txt lista\ de\ la\ compra\ Pepe.txt 2> /dev/null
| sort -u |wc -l > num\ \compra.txt
```

Con el comando `cat` concatenamos los dos archivos, luego con `2> /dev/null` en caso de error guardamos la salida ahí , hacemos un pipe para que con `sort -u` cojamos las líneas únicas, hacemos otro pipe con `wc -l` para contar el número de líneas, y por último usamos `> num\ \compra.txt` para redirigir la salida al archivo.

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 mostrara 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", el valor de errno = 2.

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

"Permission denied", el valor de errno = 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?*

Guardamos errno en una variable y tras la llamada reasignamos el valor de errno.

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?*

Al hacer el comando `top` el proceso que ejecuta clock, ocupa un 100% de la cpu.

b) *Reescribir el programa usando sleep y volver a ejecutar top. ¿Ha cambiado algo?*

Sí, el proceso no aparece en top. No ocupa nada en el procesador.

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.*

Los hilos secundarios no les da tiempo a escribir. La salida es aleatoria ya que depende del planificador, pero en general los hilos secundarios no son capaces de escribir su mensaje.

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

Los otros hilos siguen ejecutándose aun habiendo acabado el hilo principal.

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 sería necesario añadir para que sea correcto no esperar a los hilos creados.*

```
...
pthread_detach(h1);
...
pthread_detach(h2);
```

Desligamos a los hilos del hilo principal para que sean independientes. Esto se ejecuta jsto después de crear cada hilo.

Ejercicio 6: Creación de procesos.

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

No, no se puede saber. El sistema operativo es el encargado de decidir que proceso se ejecuta primero. Esta decisión la toma en función de los recursos disponibles.

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

```
else if (pid = 0) {
   printf (“Hijo PID %jd, PPID %jd\n”, (intmax_t)getpid(), (intmax_t)getppid());
}
```

En primer lugar, debemos saber que el proceso hijo tiene pid = 0 por tanto esa será la sección de código que modificaremos. Luego tenemos que conocer las funciones getpid() que obtiene el PID del hijo y getppid() que obtiene el PPID del hijo. Estas funciones devuelven un valor tipo pid_t por tanto para representarlas en un printf se deberá hacer una conversión a intmax_t y por tanto llevará “%jd”.

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

Se corresponde al primer árbol, porque es el padre quién ejecuta el fork. Las modificaciones necesarias serían las siguientes:

```
else if (pid == 0) {
   pid = fork();
}
```

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

El padre no hace wait() suficientes veces, hace únicamente una por lo que solo a un proceso hijo se le esperará, si algún otro proceso hijo de los restantes no termina antes que el proceso padre entonces se quedará huérfano.

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

while(wait(NULL) != -1)

Con esta línea lo que hacemos es que el padre haga wait hasta que deje de tener hijos. La función wait() devuelve -1 cuando el ejecutor de la función no tiene hijos.

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?*

Se imprime "Padre:"
No es correcto en un principio, ya que cuando se crea un proceso hijo con la función fork() se duplica la memoria y en esta caso modificas la memoria del proceso hijo pero luego se imprime la del proceso padre. Estas memorias no son compartidas por ambos procesos y por tanto lo copiado en el proceso hijo no se imprimirá.

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. ¿Dónde hay que liberar la memoria, en el proceso padre, en el hijo o en ambos? ¿Por qué?*

En ambos, porque cada uno tiene su propia memoria y no la comparten entre sí.

Ejercicio 8: Ejecución de programas

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

No sucede nada extraño, se sigue ejecutando "ls". Esto es porque la función execvp ("ls", argv) pasa como argumento el primer comando a ejecutar, y un array con el resto de comandos que serán ejecutados. En este caso argv[0] nunca será ejecutado ya que "ls" es el primer comando y por tanto siempre pasará del primer valor del array.

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

execl("/usr/bin/ls", "ls", "./", (char *)NULL)

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'.

a) *El nombre del ejecutable*

"ll /proc/$PID" (si tenemos permisos) o "cat & readlink /proc/$PID/exe"

b) *El directorio actual del proceso*

cat & readlink /proc/PIDcwd

c) *La línea de comandos que se usó para lanzarlo.*

"cat /proc/$PID/cmdline"

d) *El valor de la variable de entorno LANG.*

cat /proc/$PID/environ | tr '\0' '\n'

e) *La lista de hilos del proceso*

1. cd proc /$PID/task
2. ls

Ejercicio 10: Visualización de Descriptores de Fichero.

El programa se para en ciertos momentos para esperar a que el usuario pulse "enter". Se pue-den observar los descriptores de fichero del proceso en cualquiera de esos momentos si en otra terminal se inspecciona el directorio/proc/<PID>/fd, donde<PID>es el identificador del proceso. A continuación, se indica qué hacer en cada momento.

a) *Stop 1. Inspeccionar los descriptores de fichero del proceso. ¿Qué descriptores de fichero se encuentran abiertos? ¿A qué tipo de fichero apuntan?*

Para saber el tipo de descriptor usamos "ll fd". En algunos casos puede ser necesario emplear "ls –l fd".
Stop 1. Tenemos abierto el 0, 1 y 2. Son de tipo /dev/pts/3

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

Stop 2. Se añade el descriptor 3. Es de tipo txt. /home/kevin/Documentos/SOPER_2021/LAB_1/P1/file1.txt
Stop 3. Se añade el descriptor 4. Es de tipo txt. /home/marcos/SOPER/P1/file2.txt

(Se han mostrado ejemplos de los directorios en los que se encuentra para cada alumno.)

c) *Stop 4. ¿Se ha borrado de disco el ficheroFILE1? ¿Por qué? ¿Se sigue pudiendo acceder al fichero a través del directorio/proc? ¿Hay, por tanto, alguna forma sencilla de recuperar los datos?*

No se ha hecho close, por lo que no se ha borrado, se sigue pudiendo acceder mediante esa carpeta.

d) *Stop 5, Stop 6 y Stop 7. ¿Qué cambios se han producido en la tabla de descriptores de fichero? ¿Qué se puede deducir sobre la numeración de un descriptor de fichero obtenido tras una llamada a open?*

Stop 5. Ahora sí que se ha borrado completamente. Se ha hecho el close.
Stop 6. Se genera un nuevo archivo y se reutiliza el descriptor 3
Stop 7. Se genera un nuevo archivo y se crea el descriptor 5

Se puede deducir que sigue una enumeración estática ya que al eliminar el fichero ocupando la enumeración 3 observamos que el fichero de la enumeración 4 se mantiene en su lugar y más tarde con nuevos archivos la enumeración 3 es ocupada.

Ejercicio 11: Problemas con el buffer.

a) *¿Cuántas veces se escribe el mensaje "Yo soy tu padre" por pantalla? ¿Por qué?*

Se imprime dos veces ya que al hacer fork el proceso hijo tiene una copia del buffer.

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

No, porque el buffer está vacío a la hora de hacer el fork, por lo tanto, el hijo imprime “Nooo” y el padre “Yo soy tu padre”. El hecho de poner \n obliga al buffer a vaciarse.

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

Se imprime el \n pero el hijo imprime ambas strings, esto es porque el \n no fuerza el vaciado del buffer cuando se usa una variable FILE.

d) *Indicar en la memoria cómo se puede corregir definitivamente este problema sin dejar de usar printf*

Usando fflush antes de crear al hijo.

Ejercicio 12: Ejemplo de Tuberías.

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

He recibido el string: Hola a todos!
He escrito en el pipe

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

El padre se queda indefinitivamente esperando a una respuesta del hijo ya que la parte de escritura no está cerrada, la función read “bloquea” al padre hasta que llegue una respuesta.

Ejercicio 13: Shell. Escribir un programa en C (“proc_shell.c”) que implemente una shell sencilla (sin redirecciones ni estructuras de control). El ejercicio se divide en cuatro partes diferenciadas.

c) *Análisis de ejecución:*

- *Explicar qué función de la familia exec se ha usado y por qué. ¿Podría haberse usado otra? ¿Por qué?*

Hemos usado la función `execvp` ya que no hace falta que pongamos la dirección entera del comando. Por ejemplo, al usar `ls`, si usásemos `execv` tendríamos que poner `/bin/ls`, esto responde a la siguiente pregunta, sí, sí se puede usar otra, también tenemos otras posibilidades que actúan como `execve`.

- *Ejecutar con la shell implementada el comando sh -c inexistente. ¿Qué contiene la cadena con el resultado de finalización?*

`sh -c inexistente` tiene como salida: `sh: 1: inexistente: not found`

- *Hacer un programa en C que finalice llamando a abort y ejecutarlo con la shell implementada. ¿Qué contiene la cadena en este otro caso?*
  Terminated by signal 6