INFORME PRÁCTICA 1
SOPER

Álvaro Castillo García

Pablo Ernesto Soëtard

Universidad Autónoma de Madrid

Escuela Politécnica Superior

Ingeniería Informática bilingüe, grupo 2292

Sistemas Operativos

ÍNDICE

EJERCICIO 1

a)

El comando usado es: man -k pthread. La información que muestra el comando en la terminal es la siguiente:

pthread_attr_destroy (3) - initialize and destroy thread attributes object
pthread_attr_getaffinity_np (3) - set/get CPU affinity attribute in thread at...
pthread_attr_getdetachstate (3) - set/get detach state attribute in thread at...
pthread_attr_getguardsize (3) - set/get guard size attribute in thread attrib...
pthread_attr_getinheritsched (3) - set/get inherit-scheduler attribute in thr...
pthread_attr_getschedparam (3) - set/get scheduling parameter attributes in t...
pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in thre...
pthread_attr_getscope (3) - set/get contention scope attribute in thread attr...
pthread_attr_getstack (3) - set/get stack attributes in thread attributes object
pthread_attr_getstackaddr (3) - set/get stack address attribute in thread att...
pthread_attr_getstacksize (3) - set/get stack size attribute in thread attrib...
pthread_attr_init (3) - initialize and destroy thread attributes object
pthread_attr_setaffinity_np (3) - set/get CPU affinity attribute in thread at...
pthread_attr_setdetachstate (3) - set/get detach state attribute in thread at...
pthread_attr_setguardsize (3) - set/get guard size attribute in thread attrib...
pthread_attr_setinheritsched (3) - set/get inherit-scheduler attribute in thr...
pthread_attr_setschedparam (3) - set/get scheduling parameter attributes in t...
pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in thre...
pthread_attr_setscope (3) - set/get contention scope attribute in thread attr...
pthread_attr_setstack (3) - set/get stack attributes in thread attributes object
pthread_attr_setstackaddr (3) - set/get stack address attribute in thread att...
pthread_attr_setstacksize (3) - set/get stack size attribute in thread attrib...
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)

Las “llamadas al sistema” en el manual, se encuentran con el comando: man -k syscall. La información mostrada es la siguiente:

_syscall (2) - invoking a system call without library support (OBSOLETE)
afs_syscall (2) - unimplemented system calls
nfsservctl (2) - syscall interface to kernel nfs daemon
restart_syscall (2) - restart a system call after interruption by a stop signal
syscall (2) - indirect system call
syscalls (2) - Linux system calls

La llamada write, se encuentra con los siguientes comandos: man -k write; man 2 write

EJERCICIO 2

a)

El comando usado es: grep -w -n -i molino don\ quijote.txt >> aventuras.txt

Hemos utilizado -i, para poder obtener las oraciones que tengan molino, ya sea en mayúscula o minúscula. El -w, se utiliza para que sólo aparezcan las coincidencias con la palabra molino completa, y no palabras que la incluyen, como remolino. Por último está -n, que lo usamos, para que nos diga el número de línea dónde se encuentra cada palabra. Para añadirlo al final del fichero aventuras.txt redirigimos la salida (>>) del comando grep hacia el fichero aventuras.txt.

b)

El comando usado es: ls -p | grep -v / | wc -l

Primero listamos todos los ficheros y directorios que haya y les añadimos una / al final a aquellos que sean directorios con ls -p. Tras ello, seleccionamos sólo los ficheros con la opción -v / en el comando grep, conseguimos que no cuente con aquellos objetos que sean directorios. Por último, contamos cuántas líneas hay en esa lista, ya que cada fichero se encontrará en una línea distinta con wc -l.

c)

El comando usado es: cat lista\ de\ la\ compra\ Pepe.txt lista\ de\ la\ compra\ Elena.txt 2> /dev/null | sort | uniq -c | wc -l > num\ compra.txt

Con el comando cat concatenamos los archivos “lista de la compra Pepe.txt” y “lista de la compra Elena.txt”, en caso de que este comando devuelva un error, redirigimos la salida de errores a /dev/null con 2>. A la salida del comando de concatenación le hacemos un pipeline a sort, para tener ordenados alfabéticamente los elementos, y realizamos otro pipeline a uniq -c que nos devolverá solo las líneas que no se repitan. A esto le hacemos de nuevo un pipeline a wc -l que nos contará el número de líneas. Por último redirigimos la salida de lo anterior al archivo “num compra.txt” con >.

d)

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

Con ps -A -L obtenemos todos los procesos con los hilos de cada uno, esto lo pasamos por un pipeline a awk '{print $1}' que nos selecciona solo la primera columna del comando anterior, que contiene a cada proceso tantas veces como hilos tiene. Por último realizamos otro pipeline y contamos cuantas veces se repite cada proceso para saber cuanto hilos tiene con uniq -c y redirigimos la salida de este comando a “hilos.txt” con >.

EJERCICIO 3

a)

Si se intenta abrir un fichero inexistente aparece el valor de errno 2 y cómo mensaje se imprime *No such file or directory*

b)

Si se intenta abrir el fichero etc/shadow, aparece *Permission denied* y el código de error 13. Si ejecutamos el programa como administrador (sudo) aparece *Success* y error número 0.

c)

Tras haber realizado fopen se debería guardar el valor de la variable errno en un int y usarlo más tarde en la llamada a perror.

EJERCICIO 4

a)

Se puede observar como el proceso está utilizando recursos de la CPU mientras este se encuentra realizando la espera.

b)

Cuando el programa utiliza sleep en vez de clock, este ya no consume recursos de la CPU mientras se encuentra realizando la espera.

EJERCICIO 5

a)

Si el proceso no espera a los hilos y termina antes que ellos, mata a los hilos automáticamente sin que terminen de ejecutarse.

b)

Si cambiamos exit por pthread_exit, comprobamos que cuando termina el proceso principal, los hilos no mueren, continúan ejecutándose hasta acabar su programa.

c)

Simplemente antes de llamar a la función *pthread_exit()* debemos escribir:

pthread_detach(h1);
pthread_detach(h2);

para que los hilos queden desligados del proceso principal, y así acaben satisfactoriamente.
Otra opción sería desligarlos directamente cuando se llama a la función *pthread_create()* , incluyendo en los atributos: *pthread_attr_set-detachstate* .

EJERCICIO 6

El programa que describe este ejercicio se encuentra en el archivo *ejercicio_hilos.c*

Se usan las librerias standard: <stdlib.h> y <stdio.h>. Por otra parte encontramos <unistd.h> y <pthread.h> para poder crear hilos y coordinarlos con el proceso padre. También está incluida <string.h> que permitirá imprimir en la salida de error las frases escritas.

Hemos creado un archivo llamado *funciones_hilos.c,* con su correspondiente header (f*unciones_hilos.h*), en el que incluimos la función que tienen que realizar todos los hilos(*void * CalculateCube(void * arg)*) y la estructura (*Params*) en la cuál se guarda el tiempo aleatorio de espera y el valor de x.

El número aleatorio se crea con la función rand().

EJERCICIO 7

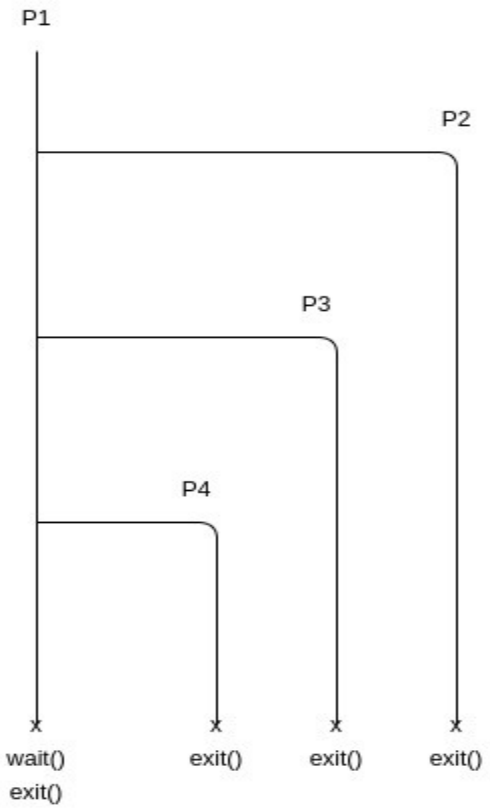
a)

No, no se puede saber en que orden se imprimirá, ya que no sabemos el orden de ejecución del planificador

b)

Simplemente sería necesario guardar en una variable el id del padre original: *ppid=getpid();* y en cada hijo llamar a la función *getpid()* e imprimir: *printf(" Hijo con pid: %d , su padre es: %d \n " , getpid(), ppid);*

c)



El árbol es así, ya que el padre original, crea tres procesos hijos, uno tras otro, y cuando estos tres terminan de ejecutarse mueren respectivamente. El padre muere cuando uno de los hijos haya acabado, es decir, solo espera a que uno de los hijos muera.

d)

Sí, deja hijos huérfanos, ya que el wait está tras el bucle for, lo que significa que esperará que termine uno de los procesos, pero no a los tres que crea.

e)

Simplemente habría que introducir la llamada a wait() dentro del bucle, para que espere secuencialmente a cada uno de los hijos. Esta solución no tendría mucho sentido, ya que el fin de crear procesos hijos es tenerlos en paralelo. Si queremos que se cumpla esta condición, simplemente debemos programar un bucle for que cuente hasta 3 y escribir en su interior la sentencia wait(), para que espere a los 3 procesos hijos.

EJERCICIO 8

El programa que describe este ejercicio se encuentra en el fichero *ejercicio_arbol.c*

Para cambiar el número de hijos que se generan, es necesario abrir el archivo y sustituir la cifra que se encuentra en: #define NUM_PROC *3*

El programa usa las librerias standard: <stdlib.h> y <stdio.h> y la de manejo de errores <errno.h>. Por otra parte encontramos <unistd.h>, <sys/types.h> y <sys/wait.h> para poder crear procesos hijos y coordinarlos con el padre.

EJERCICIO 9

a)

Cuando se ejecuta el código vemos que la frase copiada no se imprime. El programa no es correcto ya que tiene dos fallos. El primero es que no se libera la memoria reservada por la función calloc. El segundo es que la sentencia copiada se encuentra en el proceso hijo y como el padre y el hijo no comparten memoria, sino que cada uno tiene la suya, el padre no tiene acceso a la frase que el hijo ha copiado y por tanto no la puede imprimir. Por eso vemos que la salida del programa es Padre:

b)

Debemos escribir *free(sentence);* dos veces. Una en el proceso padre y otra en el hijo, ya que ambas contienen el espacio reservado para *sentence* en dos pilas de memoria distintas.
El código sería el siguiente:

```
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

a)

El programa que describe este apartado se encuentra en el archivo *ejercicio_shell.c*

Para realizar el ejercicio se han usado las librerias standard: <stdlib.h> y <stdio.h> y la de manejo de errores <errno.h>. Por otra parte encontramos <unistd.h>, <sys/types.h> y <sys/wait.h> para

poder crear procesos hijos y coordinarlos con el padre. Por último está <string.h>, que es usada para manejar cadenas de caracteres.

b)

Se ha decidido utilizar la función execvp, la v nos permite pasarle los argumentos como un array, en vez de una enumeración, esto es idóneo ya que la función strtok, previamente usada para parchear stdin, nos almacena cada argumento en un elemento de un array. También se usa p ya que nos permite utilizar las variables de entorno PATH en vez de tener que dar la ruta exacta del ejecutable del programa. Si se supiera el número de argumentos máximo que tendrán todos los comandos, se podría haber utilizado las funciones exec que contienen "l", al igual que si se hubiese querido modificar las variables de entorno, se podría haber utilizado aquellas funciones que contienen la letra "e", en caso de que lo anterior no se cumpliese, no se podría utilizar otra de la funciones exec.

c)

Al ejecutar "sh -c inexistente" la shell devuelve "sh: 1: inexistente: not found", este es el mismo mensaje que muestra bash si ejecutamos en él ese comando. Además podemos ver que el comando terminó por el siguiente motivo "Exited with value Key has expired".

d)

Al ejecutar un programa que llama a abort, la shell nos indica que el proceso acabó por la señal "Terminated by signal No such device or address".

e)

El programa que describe este apartado se encuentra en el archivo *ejercicio_shell_spawn.c*

Contiene las mismas librerias que el ejercicio_shell.c, pero se le ha añadido <spawn.h> para poder usar la función de posix_spawnp()

EJERCICIO 11

Hemos utilizado la carpeta del proceso 1

a)

El fichero que contiene el nombre del ejecutable es *cmdline*. La información que se muestra es: /sbin/init, lo que significa que el nombre del ejecutable será init.

b)

El fichero usado para ver el directorio actual del proceos es *cmdline*. El directorio actual del proceso es: /sbin/init

c)

Para ver la línea de comando que mandó el proceso, hay que examinar el fichero *comm*. La información que se muestra es: systemd.

d)

Para este apartado, desde el proceso init, no es posible acceder al fichero, así que se ha utilizado el proceso 1855. Para ver las variables de entorno, se usa el fichero *environ*. La información que se muestra en referencia a la variable LANG es: LANG=es_ES.UTF-8

e)

Podemos ver el número de hilos en el fichero *status*. Si queremos saber la lista de hilos, esta se encuentra en el directorio *task*. La información que se muestra son ficheros de los diferentes hilos.

EJERCICIO 12

a)

Al dirigirse a la ruta /*proc*/<PID>/fd se puede observar como existen 3 descriptores de ficheros, el 0, 1 y 2. Estos son la entrada estándar, la salida estándar y la salida de errores, respectivamente. Los ficheros son de tipo lrwx, esto es que se pueden linkear, leer, escribir y ejecutar.

b)

Al llegar a Stop 2, podemos ver que se ha creado un nuevo descriptor de fichero, el 3, que corresponde con file1.txt, y al legar a Stop 3, podemos ver que también se ha creado un nuevo descriptor de fichero, el 4, que se corresponde con file2.txt.

c)

Al llegar a Stop 4, se puede observar como el fichero file1.txt ha sido borrado del disco, esto se debe a que se ha realizado un unlink, y al no estar el fichero referenciado por ningúna otra ruta (ya que lo acabamos de crear), este ha sido borrado del disco.
Pese a esto, en el directorio /*proc*/<PID>/fd se sigue pudiendo observar el descriptor del fichero, pero esta vez en rojo. Esto se debe a que seguimos teniendo el descriptor de fichero de ese archivo en la variable file1, por lo que se podrían recuperar los datos volviendo a abrir una ruta sobre ese descriptor de fichero.

d)

Al llegar al Stop 5, el descriptor 3 es borrado de /*proc*/<PID>/fd, en el Stop 6, un nuevo descriptor de fichero es creado y este ocupa el lugar del 3 anteriormente borrado. Por último, en el Stop 7, un nuevo descriptor de archivo es inicializado sobre el mismo archivo que el 3, pero con diferentes flags, y a este se le adjudica el número 5.

Se puede observar como la asignación de descriptores de fichero es ascendente, y siempre que quede algún hueco en la tabla de descriptores, este será rellenado con uno nuevo cuando sea necesario.

EJERCICIO 13

a)

La frase: Yo soy tu padre, se imprime dos veces. Esto se debe a que el primer “Yo soy tu padre” se almacena en el buffer pero no llega a imprimirse antes de que se realice el fork. En el proceso hijo se añade “Noooo” al buffer, por lo que quedaría así “Yo soy tu padreNoooo” ya que el buffer contenía el “Yo soy tu padre” anterior (se copia la memoria de un proceso a otro). Al realizarse el exit el buffer se ve forzado a vaciarse, por lo que se imprime “Yo soy tu padreNoooo”, y por último, el buffer del proceso padre, que contenía “Yo soy tu padre” se vacía, observándose por pantalla el mensaje “Yo soy tu padreNoooooYo soy tu padre”.

b)

Tras poner el separador de línea ya sólo se imprime una vez. Esto se debe a que al introducir \n al final de cada mensaje, el buffer se ve forzado a vaciarse (flush) por lo que al realizar el fork el buffer ya está vacio y solo contendrá el “Noooo” cuando el hijo imprima, por lo que no se duplicará el “Yo soy tu padre”.

c)

Si se redirige la salida a un fichero, de nuevo se imprime dos veces la misma frase. En este caso, pese a que se ha realizado un \n, el fichero contiene dos veces “Yo soy tu padre”, ya que el sistema operativo mantiene en caché un buffer para minimizar las llamadas a entrada/salida, que son muy lentas. Por lo que vuelve a pasar lo mismo que en el apartado a), pero esta vez por el buffer del caché del sistema operativo.

d)

El problema se corregirá usando la función fflush(stdout) tras llamar al printf(“Yo soy tu padre”), para que se vacíe el buffer.

EJERCICIO 14

a)

Dependiendo de la velocidad de ejecución de los procesos, por pantalla se puede ver impreso:
He recibido el string: Hola a todos!
He escrito en el pipe

ó:

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

b)

Si el padre no cierra el extremo de escritura, el programa se queda colgado esperando un mensaje de este hacia el hijo. Para que el pipe se elimine, es necesario que se cierre la parte de escritura con la intención de mandar el mensaje de final de fichero al proceso lector.

c)

No se imprime nada en la pantalla, por dos motivos. El primero es que al haber puesto un retraso en el hijo y al haber quitado la sentencia wait() del padre, lo más probable es que el proceso padre acabe antes que el hijo dejándolo huérfano y por tanto, se cierre el pipe automáticamente. Por otra parte al haber quitado la sentencia para leer la tubería desde el padre, este no imprimirá en pantalla :He recibido el string: [string].

EJERCICIO 15

El programa que describe este ejercicio se encuentra en el archivo *ejercicio_pipes.c*

Hemos creado un archivo llamado *funciones_pipes.c,* con su correspondiente header (f*unciones_pipes.h*), en el que incluimos las funciones para mandar un mensaje a traves del pipe (*int SendMessage(int fd, char * message)*), para recibirlo (*int ReadMessage(int fd, char * message)*) y para escribir un mensaje en el descriptor de fichero pedido (*int WriteOnFile(char * message, char * filename)*).

Para realizar el ejercicio se han usado las librerias standard: <stdlib.h> y <stdio.h> y la de manejo de errores <errno.h>. Por otra parte encontramos <unistd.h>, <sys/types.h> y <sys/wait.h> para poder crear procesos hijos y coordinarlos con el padre.

Para generar un número aleatorio hemos decidido utilizar la función rand() con la condición de srand(time(NULL)), y por ello hemos incluído <time.h>. Tambíen se usan funciones de <string.h> como strlen() para conseguir saber la longitud del mensaje mandado. Por último, las librerias <fcntl.h> y <sys/stat.h> son necesarias para utilizar los descriptores de fichero y escribir en numero_leido.txt.