Sistemas Operativos. Práctica 1.

Pablo Cuesta Sierra y Álvaro Zamanillo Sáez

Semana 1

Ejercicio 1

a) Para buscar las funciones relacionadas con hilos, buscamos todas aquellas que contengan "pthread" usando la opción -k del comando man. El comando a usar es:

```
$ man -k pthread
```

Lista de funciones resultante:

```
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 el manual en qué sección se encuentran las llamadas a sistema y buscar infromación sobre write:

```
$ man man
```

Con este comando averiguamos que la sección relacionada a *system calls* es la 2. Por lo tanto usamos el comando:

```
$ man 2 write
```

Ejercicio 2

a) El comando empleado es:

```
$ grep -w molino "don quijote.txt" >> aventuras.txt
```

Usamos el comando grep para buscar las apariciones de "molino". Como queremos que sea la palabra *molino* y no el grama *molino*, añadimos la opción `-w`. Finalmente redireccionamos la salida con `>>` en vez de `>` para que se añada al final del fichero, en lugar de reemplazarlo.

Nota: podríamos utilizar también la opción `-i` de `grep` para que nos salieran también las apariciones de la palabra donde empieza con mayúscula.

b) El *pipeline* es el siguiente:

```
$ ls | wc -l
```

La salida de `ls` es una lista (fichero) con los ficheros del actual directorio. Esta lista la usamos como input del comando wc, que acompañado de la opción -l, cuenta las lineas de dicha lista.

c) En este caso el *pipeline* es:

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

Primero concatenamos los dos ficheros (redirigiendo el mensaje error, en caso de haberlo). Después, dirigimos la salida para que sea el input de `sort`, para que luego `uniq` quite las líneas repetidas. Finalmente, contamos el número de líneas distintas con `wc -l` y dirigimos la salida al fichero `"num compra.txt"`.

Ejercicio 3

a) Si intentamos abrir un fichero inexistente recibimos el mensanje `"No such file or directory"`. El código errno asociado es 2.

b) En el caso de intentar abrir el fichero `/etc/shadow` el mensaje de error es `"Permission denied"`, que corresponde al valor de errno 13.

c) Justo después de la instrucción `fopen()` se debería guardar el valor de errno en otra variable ya que la llamada a `printf()` podría modificar la variable global errno.

```
pf=fopen(argv[1],"r");
x=errno;
printf("El valor de errno es %i",x);
errno=x; //Aquí aseguramos que la función perror() va a imprimir el código
         //de error de fopen ya que hemos restaurado el valor de errno
         //asociado a fopen()
perror("fopen");
```

Ejercicio 4

a) Durante los 10 segundos de espera el proceso asociado al programa está en estado "R"(runnable), es decir se está ejecutando (intercalándose con otros).

b) En este caso el proceso está en estado "S" (interruptible sleep) o lo que es lo mismo, esperando a un suceso (el fin de la espera marcada por sleep()), mientras se ejecutan otros procesos.

Ejercicio 5

a) Si el proceso no hubiese esperado a los hilos, cuando el hilo principal llegase a `exit()`, el programa finalizaría matando los hilos que no tendrían por qué haber acabado. De hecho, si quitamos del código las instrucciones `pthread_join`, vemos que cada hilo solo tiene tiempo de escribir una letra antes de que el hilo principal llegue al exit y finalice, por tanto, la ejecución de todos los hilos.

b) En este caso, ocurre exactamente lo mismo que en anterior (se imprimen las dos primeras letras–una de cada hilo–, luego se imprime el mensaje de la línea 51 en el código dado). Sin embargo, después de esto, continúan ejecutándose los dos hilos hasta que cada uno de ellos llega a su función `exit()`.

Es decir, `pthread_exit()` termina la ejecución del hilo principal, pero permite que se terminen todos los hilos que del proceso antes de terminarlo. Lo cual no pasa si se llama directamente a `exit()`, como hemos visto en el apratado 5a).

c) Para que se espere la terminación de los hilos sin que el hilo principal ocupe al procesador, pero sin utilizar ni `pthread_exit()` ni `pthread_join()`, podemos hacer lo siguiente:

```
1  #define max(a,b) ((a)>(b)?(a):(b))
2
3  int _h_count=0;
4
5  void *slow_printf(void *arg) {
6      const  char *msg = arg;
7      size_t i;
8
9      for (i = 0; i < strlen(msg); i++) {
10         printf(" %c ", msg[i]);
11         fflush(stdout);
12         sleep (1);
13     }
14     _h_count++; //para saber cuántos han terminado (0, 1, o 2)
15     return  NULL;
16 }
```

```
17
18 int main(int argc, char **argv){
19     pthread_t h1, h2;
20     char *hola="Hola", *mundo="Mundo";
21     int error;
22
23     error = pthread_create(&h1, NULL, slow_printf, hola);
24     if(error != 0){
25         fprintf(stderr, "pthread_create: %s\n", strerror(error));
26         exit (EXIT_FAILURE);
27     }
28     error = pthread_create(&h2, NULL, slow_printf, mundo);
29     if(error != 0){
30         fprintf(stderr, "pthread_create: %s\n", strerror(error));
31         exit (EXIT_FAILURE);
32     }
33     error = pthread_detach(h1);
34     if(error != 0){
35         fprintf(stderr, "pthread_detach: %s\n", strerror(error));
36         exit (EXIT_FAILURE);
37     }
38     error = pthread_detach(h2);
39     if(error != 0){
40         fprintf(stderr, "pthread_detach: %s\n", strerror(error));
41         exit (EXIT_FAILURE);
42     }
43
44     sleep(max(strlen(hola), strlen(mundo)));
45     while (_h_count != 2);/*nos aseguramos de que han terminado*/
46     printf("El programa %s termino correctamente\n", argv[0]);
47     exit(EXIT_SUCCESS);
48 }
```

Si no llamamos a `pthread_join()`, tenemos que desilgar los hilos (llamando a `pthread_detach()`).

Además, guardamos una variable global que cuenta los que hilos han terminado, y por la orden: `while(_h_count!=2);`, nos aseguramos de que el hilo principal no termina antes de que los otros dos hayan terminado. Para evitar que el hilo principal esté comprobando esta variable cuando sabemos que no pueden haber terminado, llamamos justo antes a `sleep()`, para que el hilo principal se quede esperando un suceso (que el tiempo máximo que tarda uno de los hilos pase), en lugar de estar ejecutándose sin sentido.

Semana 2

Ejercicio 6

a) No se puede saber ya que después de cada `fork()` el planificador puede establecer un orden de ejecución distinto de los cuatro procesos que hay en total.

b) Como queremos que sea el hijo el que imprima su PID y el de su padre, hay que modificar el bloque if correspondiente al hijo (`fork()` devuelve 0 en el hijo). En el printf llamamos a las funciones `getpid()` y `gepppid()` haciendo cast de sus valores de retorno:

```
1     else if(pid==0){
2         printf("Hijo: pid=%jd, ppid=%jd\n", (intmax_t)getpid(), (intmax_t)getppid());
3         exit(EXIT_SUCCESS);
4     }
```

Esta opción es posible que no imprima correctamente el pid del padre (si el proceso se queda huérfano). Sin embargo, si antes de llamar a `fork()` guardamos en una varibale local el pid del padre: `ppid=getpid();`, nos aseguramos que todos los hijos tienen guardado el pid del padre y se puede imprimir de esta manera.

c) El código se corresponde al diagrama (a); ya que los hijos depués de imprimir finalizan (llamada a `exit()`) y en cambio el padre vuelve a crear un nuevo hijo en la siguiente iteración. Para obtener el otro diagrama habría que modificar los bloques del hijo y del padre: el proceso padre es el que llama a `exit()` y el hijo no lo hace, para crear un nuevo proceso en la siguiente iteración del bucle:

```
1     else if(pid==0){
2         printf("Hijo: pid=%jd, ppid=%jd\n", (intmax_t)getpid(), (intmax_t)getppid());
3     }
```

```
    else if(pid > 0) {
        printf("Padre %d\n", i);
        exit(EXIT_SUCCESS);
    }
```

d) El código original puede dejar huérfanos ya que nada asegura que los 3 hijos vayan a finalizar antes que el padre. En cuanto el proceso padre se encuentre en la línea de ejecución del `wait()` y un hijo suyo finalice (o haya finalizado), el proceso padre finalizará también dejando huérfano a los posibles hijos restantes (máximo 2). Es decir, `wait()`, usado como en el código dado, tan solo espera al primer hijo que finalice.

e) Una forma de evitar esto es cambiar esa línea de código por: la línea: `while(wait(NULL)!=-1);`. El padre esperará a un hijo hasta que no haya procesos hijos que esperar (*no unwaited-for children*), cuando devolverá -1 y podrá finalizar el padre.

Ejercicio 7

a) Al ejecutar vemos que la cadena *sentence* que imprime el padre está vacía. El error del programa es no tener en cuenta que al crear el proceso hijo, éste tiene una memoria separada del padre por lo que los cambios que se hagan desde el hijo no son visibles desde el padre. Cada puntero *sentence* apunta a una dirección de memoria distinta. (debido al *Copy On Write* podría ser que si no se trata de modificar la información de ese bloque de memoria no se llegase a crear una copia. No obstante en este caso no hay dudas ya que estamos modificando la memoria apuntada por *sentence* al llamar a la función `strcpy()`).

b) Habría que liberar memoria tanto en el padre como en el hijo ya que se está creando una nueva copia del bloque de memoria alocado en el proceso padre como hemos explicado en el apartado anterior:

```
    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 8

a) El resultado es el mismo. Según el manual de execvp, el hecho de que el primer elemento en el array de argumentos que se pasa (`const char *argv[]`) sea el nombre del fichero asociado al fichero que se va a ejecutar, es tan solo una convención. Sin embargo, no cambia el resultado, ya que se ejecuta el fichero que se indica en el primer argumento de la llamada a `execvp()`.

b) Tenemos que cambiar la orden de llamada a `execvp` por lo siguiente:

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

Debido a que en este caso la función no contiene la letra `p` tras el sufijo `exec`, es necesario especificar la ubicación del fichero a ser ejecutado (en este caso: `/bin/ls`). Después, incluimos los argumentos de la función. En este caso, debido a la `l` después del prefijo `exec`, éstos no deben pasarse como un array de cadenas, sino uno a uno (las funciones que tienen el prefijo `execl` son funciones variádicas). Como antes, por convención, el primero de ellos es el nombre del fichero que se va a ejecutar (`"ls"`), y luego los argumentos: en este caso, solamente `"./"`. Finalizando con un puntero a `NULL` (además, debemos hacer cast a `(char*)`).

Semana 3

Ejercicio 9

a) Nombre el ejecutable: en el fichero `/proc/`*`[pid]`*`/stat`, el segundo campo contiene entre paréntesis el nombre del ejecutable: `bash`.

b) Hay que ver a dónde apunta el link `cwd`: `/proc/6060`

c) La línea de comados que se usó para lanzarlo se encuentra en el fichero: `/proc/`*`[pid]`*`/cmdline`

Y es: `bash`.

d) Lo podemos buscar en el fichero `environ`, y es: `LANG=es_ES.UTF-8`

```
cat /proc/6060/environ | tr '\0' '\n' | grep LANG
```

e) La lista de hilos del proceso se encuentra haciendo `ls /proc/[pid]/task`, ya que este directorio es donde se encuentran los subdirectorios de los hilos. En este caso solo hay un hilo: `6060`.

Ejercicio 10

a) Encontramos los descriptores de fichero 0, 1 y 2, que corresponden respectivamente a stdin, stdout y stderr.

b) En el segundo *Stop* vemos que hay un nuevo descriptor (3) que apunta a file1.txt y en el siguiente *Stop*, otro nuevo (4) que apunta a file2.txt

c) Tras la llamada a unlink vemos que el descriptor de fichero que apuntaba a file1.txt aparece marcado como borrado –`(deleted)`– en la tabla aunque permanece ahí. Esto se debe a que solo hemos eliminado la ruta para acceder al fichero pero no lo hemos cerrado.

Se sigue podiendo acceder al fichero desde `/proc/[pid]/fd`, y podemos acceder a su contenido, por lo que se pueden recuperar los datos.

Una manera sencilla de recuperarlo (cambiando el código) después de la orden `unlink(FILE1)` sería hacer lo siguiente:

```
sprintf(file_path, "/proc/%jd/fd/%d",(intmax_t)getpid(),file1);
if(!fork())
    execlp("cp", "cp", file_path, FILE1, (char*)NULL);
```

De este modo se copia el fichero que no se ha cerrado al directorio desde el que se está ejecutando el programa, y cuando se cierra no se pierde su contenido.

d) Tras el *Stop 5*, el descriptor del fichero file1.txt ya se ha eliminado de la tabla (todas sus rutas se han eliminado con `unlink()` y el proceso lo ha cerrado con `close()`). Tras el *Stop 6*, se ha reutilizado el descriptor 3, antes asignado al fichero file1.txt, para el fichero file3.txt. Y tras el *Stop 7*, se asigna otro descriptor (5) al mismo fichero (file3.txt); por lo que ahora tenemos 0, 1, 2 como al inicio, 3 y 5 apuntando a file3.txt, y 4 apuntando a file2.txt.

Por lo que hemos visto, al abrir un nuevo fichero, se asigna el descriptor más pequeño que esté libre tras cada llamada a `open`.

Ejercicio 11

a) El mensaje "Yo soy tu padre" se imprime dos veces en pantalla porque la llamada a `fork()` se hace sin que se haya vaciado el buffer de salida. Por lo tanto el proceso hijo, cuyos datos son una copia del proceso padre, hereda una copia de este buffer de salida (que no está vacio). Cuando el hijo acaba su ejecución vacia su buffer que contiene tanto el mensaje "Yo soy tu padre" como "Noooooo". Por otro lado el padre al terminar también vuelca su buffer; en este caso solo contiene "Yo soy tu padre".

b) Si añadimos el salto de línea ya no ocurre lo mencionado anteriormente. Esto se debe a que cuando se está imprimiendo en *stdout*, el buffer está configurado para volcarse con cada salto de línea o al llenarse. Es por eso que cuando se crea el hijo, el buffer de salida que hereda está vacío.

c) Al redirigir la salida a un fichero, la configuración del buffer ya no es la del apartado anterior (vaciarse con salto de línea), sino que solo se vacía tras una llamada a `fflush(file)` o cuando se llena.

d) Para solucionar los problemas, lo más sencillo es asegurarse que el buffer se vuelca antes de crear el proceso hijo y esto se consigue en todos los casos con la llamada a `fflush(stdout)`.

Ejercicio 12

a) Tras ejecutar el programa vemos que el padre imprime por pantalla: "He recibido el string: Hola a todos!", y el hijo: "He escrito en el pipe". Como se ha comentado anteriormente, el orden de estos mensajes puede variar.

b) Si no cierra el extremo de escritura el mensaje impreso en pantalla es el mismo pero no finaliza el proceso, ya que el padre sigue esperando a una escritura en la tubería. El descriptor de fichero `f[1]` sigue abierto: por él mismo, por lo que se queda bloqueado en la orden `nbytes=read(...)`. En el anterior caso, como todos los procesos habían cerrado el fichero de salida `f[1]`, en esa orden el padre leía `EOF`, por lo que `read()` devolvía `0` y el bucle se terminaba.

Ejercicio 13

c) Hemos usado la función `execvp` ya que los argumentos del comando tras la ejecución del hilo, han sido colocados en un vector, la 'p' de la función es para que use la variable local PATH al ejecutar el comando. No se puede usar otra función, ya que no conocemos el número de argumentos que se van a escribir, por lo que tenemos que usar una función con 'v'; y por otro lado, no sabemos la ruta del comando que se quiere ejecutar.

Al ejecutar el comando `sh -c inexistente` se obtiene la cadena: `Exited with value: 127`.

Al ejecutar un programa que finaliza llamando a `abort()`, recibimos la señal Aborted (que tiene asociado el código 134). La cadena en este caso es: `Terminated by signal: Aborted`.

Sobre la implementación de la shell:

Básicamente el código está estructurado según las indicaciones del enunciado: un bucle principal en el *main* lee los comandos a ejecutar línea a línea hasta que se introduzca *EOF*. La línea leida forma parte de una estructura que se envía como argumento del hilo (hacemos cast a puntero a void). El hilo va a preparar el *array* que se usará posteriormente en la llamada a *execvp*. Para asegurar que no se lance el proceso nuevo antes de que el procesado de la línea haya finalizado, usamos la función *pthread_join*. Como la función *strtok* altera la *string* original, escribimos en la tubería la línea leída antes de la llamda al hilo. A propósito del uso de la tubería, la creamos antes de empezar a leer en el bucle y por supuesto, antes de lanzar el proceso que tiene que leer de la tubería y escrbir en el fichero log.txt para que este proceso tenga en su tabla de descriptores, los asociados a la tubería. En el proceso padre cerramos el extremo de lectura, y en el hijo, el de escritura. Por otro lado, hemos empleado la función *dprintf* para volcar la línea leída y el estado de terminación del proceso lanzado por *execvp*. El estado de terminanción lo recogemos con la función bloqueante *wait*, que va a impedir al proceso padre continuar hasta que finalice el proceso que ejecuta el comando. Una cosa a mencionar es PROBLEMA CON STRSIGNAL!!. El programa finaliza cuando se introduce EOF y el padre cierra su extremo de escritura de la tubería. El proceso en el otro extremo de la tubería finalizará entonces (tras leer todo los datos que pudiesen quedar en la tubería). Como se ve en el código, no pueden quedar procesos huérfanos ya que el proceso padre hace un *wait* por cada proceso lanzado (o lo que es equivalente, por cada llamada a la función *fork*)