Eduardo Terrés Caballero
Gregorio Blázquez Martínez

PRÁCTICA 1

Ej. 1) Uso del Manual

a) El comando empleado es "`man -k pthread`". Concretamente, "`man -k pthread > file.txt`", para poder copiar el resultado con mayor facilidad.

```
pthread_attr_destroy (3) - initialize and destroy thread attributes object
pthread_attr_getaffinity_np (3) - set/get CPU affinity attribute in thread attributes o...
pthread_attr_getdetachstate (3) - set/get detach state attribute in thread attributes o...
pthread_attr_getguardsize (3) - set/get guard size attribute in thread attributes object
pthread_attr_getinheritsched (3) - set/get inherit-scheduler attribute in thread attrib...
pthread_attr_getschedparam (3) - set/get scheduling parameter attributes in thread attr...
pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in thread attribu...
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 ob...
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 o...
pthread_attr_setdetachstate (3) - set/get detach state attribute in thread attributes o...
pthread_attr_setguardsize (3) - set/get guard size attribute in thread attributes object
pthread_attr_setinheritsched (3) - set/get inherit-scheduler attribute in thread attrib...
pthread_attr_setschedparam (3) - set/get scheduling parameter attributes in thread attr...
pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in thread attribu...
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 ob...
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...
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 ...
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 attri...
pthread_mutexattr_getrobust_np (3) - get and set the robustness attribute of a mutex at...
pthread_mutexattr_setpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_setrobust (3) - get and set the robustness attribute of a mutex attri...
pthread_mutexattr_setrobust_np (3) - get and set the robustness attribute of a mutex at...
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the thread read...
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the thread read...
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) Ejecutando el comando "`man syscalls`", se puede encontrar entre la lista de funciones de llamadas a sistema la función write:

```
       waitid(2)                2.6.10
       waitpid(2)               1.0
       write(2)                 1.0
       writev(2)                2.0
       xtensa(2)                2.6.13       Xtensa only
```

De esta forma, ejecutando el comando "`man 2 write`" se obtiene la documentación buscada.

Ej. 2) Comandos y Redireccionamiento

a) El comando utilizado es:

`grep " molino" don\ quijote.txt >> aventuras.txt`".

El programa grep permite buscar la secuencia de caracteres " molino", que incluye un espacio para evitar palabras cuyo lexema sea "molino" pero tengan prefijos (p.e. "remolino"). Seguidamente, se redirige la salida al fichero mediante el símbolo >>.

b) Los comandos utilizados son:

```
ls | wc -l
```

Con el símbolo | hacemos un pipeline que ejecute primero *ls* obteniendo una lista de los ficheros del directorio actual y toma su salida como la entrada de *wc -l*, que cuenta el número de líneas, que coincide con el número de ficheros.

c) Los comandos utilizados son:

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

Primero, con el comando *cat* se concatenan las dos listas, junto con el redireccionamiento de errores 2>/dev/null para que ignorarlos. Se emplea la salida de *cat* como entrada de *sort,* que ordena el fichero, de forma que las líneas repetidas quedan contiguas para que el comando *uniq* pueda filtrar las repeticiones. Finalmente, se cuentan las líneas de la salida mediante *wc –l*, guardando el número en "numcompra.txt" a través del símbolo de redirección >.

Ej. 3) Control de Errores

a) El mensaje de error es: *''No such file or directory''*, con código de error 2.

b) El mensaje de error es: *''Permission denied''*, con código de error 13.

c) Habría que almacenar el valor de *errno* antes de imprimirlo pues posiblemente este valor cambie incluso si no se produce otro error.

Ej. 4) Espera Activa e Inactiva

a) Aparece un proceso, con el nombre del programa, que consume el 100% de la CPU durante 10 segundos. Una vez acaba el programa, muere el proceso.

b) Ahora no hay un proceso que ocupe la CPU esos diez segundos. Como hemos visto en el *pdf*, el planificador duerme el hilo principal (y único) del proceso que ejecuta la llamada a *sleep*, de forma que no consume recursos de CPU durante ese tiempo.

Ej. 5) Finalización de Hilos

a) Al eliminar las llamadas a *pthread_join*, el hilo principal, que ejecuta el *main*, no espera a que los hilos que ha creado terminen su tarea. De esta forma, antes de que se imprima ningún carácter, el *main* ejecuta la sentencia de salida (*exit*), y ambos hilos son destruidos, independientemente del estado de la tarea que tenían asignada.

b) En este caso, se respeta la ejecución de ambos hilos. Como indica la documentación, la función *exit* es *MT-Unsafe*, de forma que su llamada en procesos multihilo puede ser problemática, como es el caso. Por contraposición, la función *pthread_exit* es *MT-Safe*, y respeta la ejecución simultánea de hilos, dejando que finalicen su tarea antes de acabar el hilo principal.

En la sección NOTAS de la documentación de esta última función se hace alusión directa a lo que pretende reflejar este apartado:

Sección NOTES, ejecución de `man pthread_exit`

```
To allow other threads to continue execution, the main thread
should terminate by calling pthread_exit() rather than exit(3).
```

c) Mediante llamadas a la función *pthread_detach*, se desligan los hilos, de forma que no serán esperados ni devolverán una variable de retorno.

Modificación *thread_example.c*

```
int main(int argc, char *argv[]) {
	pthread_t h1;
	pthread_t h2;
	char *hola = "Hola ";
	char *mundo = "Mundo";
	int error;

	error = pthread_create(&h1, NULL, slow_printf, hola);
	if (error != 0) {
		fprintf(stderr, "pthread_create: %s\n", strerror(error));
		exit(EXIT_FAILURE);
	}

	error = pthread_detach(h1);
	if (error != 0){
		fprintf(stderr, "pthread_detach: %s\n", strerror(error));
		exit(EXIT_FAILURE);
	}


	error = pthread_create(&h2, NULL, slow_printf, mundo);
	if (error != 0) {
		fprintf(stderr, "pthread_create: %s\n", strerror(error));
		exit(EXIT_FAILURE);
	}

	error = pthread_detach(h2);
	if (error != 0){
		fprintf(stderr, "pthread_detach: %s\n", strerror(error));
		exit(EXIT_FAILURE);
	}

	printf("El programa %s termino correctamente \n", argv[0]);
	pthread_exit(NULL);
}
```

Ej. 6) Creación de Procesos

a) No es posible predecir el orden de impresión. Esto se debe a que la llamada a *wait* es posterior a la impresión de los datos, de forma que una vez sucede la bifurcación, dependerá del sistema operativo el orden de ejecución de las llamadas.

Únicamente se puede afirmar que la impresión de “Padre <int>” estará ordenada, de forma que primero se imprimirá la correspondiente a 0, luego a 1 y finalmente a 2. No obstante, es posible que se intercalen impresiones realizadas por cualquiera de los hijos. Este orden relativo de impresión de los padres se debe a que es el hilo principal del proceso inicial el encargado de ejecutar las tres iteraciones de bucle, que se dará en orden por la naturaleza secuencial de la ejecución.

Eduardo Terrés Caballero
Gregorio Blázquez Martínez

b) Se almacena el valor del ID del proceso padre en una variable local para evitar recibir el ID de un posible proceso que haya recogido al hijo en caso de haber quedado huérfano.

```
int main(void) {
	int i;
	pid_t pid;
	intmax_t ppid;

	for (i = 0; i < NUM_PROC; i++) {

		//igual

		else if (pid == 0) {
			printf("Hijo %d\tPID: %jd\tPPID: %jd\n", i,(intmax_t)
			getpid(),(intmax_t) ppid);
			exit(EXIT_SUCCESS);
		}

		//igual
}
```

c) Al primero, pues es el proceso original P1 el que llama tres veces a *fork* creando tres hijos, que posteriormente hacen un exit (mueren) justo después de la impresión de información.

```
int main(void) {
	static int i;
	pid_t pid;
	intmax_t ppid;

	if(i == 3) exit(EXIT_SUCCESS);
	else i++;

	ppid = getpid();
	pid = fork();

	if (pid <  0) {
		perror("fork");
		exit(EXIT_FAILURE);
	}

	else if (pid == 0) {
		printf("Hijo %d\tPID: %jd\tPPID: %jd\n", i,(intmax_t)
		getpid(),(intmax_t) ppid);
		main();
		exit(EXIT_SUCCESS);
	}
```

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

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

d) Porque el proceso original que genera a los tres hijos hace un único *wait*, por lo que no recoge a dos de sus tres procesos hijos.

e) Tenemos dos opciones:

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

De esta manera añadiendo solo una línea en el *if* del padre del bucle y quitando el *wait* del final podemos hacer que cada vez que el padre cree un hijo espere a que este acabe. De esta manera podemos acercarnos más a predecir lo que va a suceder (aunque no sepamos si se va a imprimir primero la línea del padre o del hijo, sabemos que no se va a crear otro hijo hasta que el anterior haya acabado).

Otra forma sería que el proceso inicial espere a los tres hijos antes de su finalización:

```
for(i = 0; i < NUM_PROC; i++) wait(NULL);
exit(EXIT_SUCCESS);
```

Ej. 7) Espacio de Memoria

a) Cuando se ejecuta el código se imprime “Padre: ”, mostrando que la cadena *sentence* del padre está igual que después de ser inicializada por el *calloc,* ya que las modificaciones en la variable *sentence* por parte del hijo no afectan al proceso padre – se trata de espacios de memoria independientes -. Así pues, el programa es incorrecto. Además, no se libera la memoria reservada para la cadena de caracteres.

b) Es necesario liberar la cadena *sentence* tanto en el proceso padre como en el hijo, pues ambos tienen su memoria reservada para esa cadena y ambos necesitan liberarla.

```
int main(void) {
      // igual
      free(sentence);
}
```

Ej. 8) Ejecución de programas

a) No sucede nada, pues el primer argumento que le pasamos en *argv* es el nombre del programa (por defecto sería *ls*) y este no presenta ninguna utilidad de cara a la ejecución, sino que suele ser empleado para poder hacer referencia al programa en mensajes de error o avisos.

b) Para utilizar la función *execl* hay que introducir los parámetros del programa a ejecutar como variables separadas de entrada, incluyendo el *path* como primer argumento y *NULL* en última posición. El array *argv* ya no es necesario.

```
if (execl("/usr/bin/ls","ls", "./", (char*) NULL)) {
            perror("execvp");
            exit(EXIT_FAILURE);
        }
```

Ej. 9) Directorio de Información de Procesos

En el proceso de la terminal empleada – acceso mediante *self* -:

a) El nombre del ejecutable es *bash,* consultando en el fichero *status.*

b) La dirección del proceso es */proc/self* o, análogamente, */proc/2806.*

c) La línea de comandos que se empleó fue *bash.* Esta información se almacena en el fichero *cmdline.*

d) En el fichero *environ*, se encuentra el valor de LANG = en_US.UTF-8.

e) En la carpeta *task*, se encuentra un directorio por cada hilo que conforma el proceso.

Ej. 10) Visualización de Descriptores de Fichero

a) Stop 1: hay tres descriptores abiertos – 0, 1, 2 -.

b) Stop2: se genera otro descriptor de fichero, de valor 3, asociado a *file1*. Stop 3: se genera un quinto descriptor, con valor 4, asociado a *file2*.

c) Stop 4: no se ha borrado el fichero de disco, aunque aparece marcado. Se debe a que el proceso todavía mantiene abierto el fichero. El contenido del fichero es accesible mediante la carpeta */proc*, y se podría recuperar restaurando el *link*. Mediante *cat,head,tail*, o bien programas como *gedit* se puede acceder a la información de manera sencilla.

d) Tras el Stop 5 y la ejecución de la sentencia *close*, se elimina el descriptor de fichero *file1*. Stop 6: se crea un nuevo descriptor con valor 3 - pues es el numero entero más bajo disponible -, asociado a *file3*. Stop 7: se crea un sexto descriptor, de valor 5, asociado a *file4.*

La numeración de los descriptores de ficheros parece ser el entero positivo más bajo disponible.

Ej. 11) Problemas con el Buffer

a) Se imprime 2 veces, puesto que al hacer el *fork* se duplica la memoria y el buffer, al que aún no le había llegado ningún '\n', ni se había llenado, aún almacenaba "Yo soy tu padre". De esta manera, tras el *fork* tenemos dos buffers con la cadena, que terminan imprimiéndose.

b) Al añadir el carácter de nueva línea, se arregla el error, pues el buffer de salida se vacía antes del *fork*. Tras la bifurcación, se duplica un buffer vacío.

c) Al redirigir la salida a un fichero, los caracteres pasan a almacenarse en el buffer reservado para el objeto de tipo *FILE*, que tiene una implementación de cara al vaciado del buffer distinta a *stdout*. En estas condiciones, tras el *fork*, se duplica el buffer de nuevo, aun incluyendo '\n' se imprime la frase dos veces.

d) Para solucionar el problema, se puede añadir una llamada a *fflush* después de *print*f, de forma que el buffer queda vacío previo al *fork*.

Ej. 12) Ejemplo de tuberías

a) El resultado de la ejecución es:

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

b) Que el programa no finaliza tras imprimir lo mismo que antes. Esto se debe a que al no cerrar el descriptor de escritura el padre puede leer y escribir en la tubería. Una vez termina la escrita el hijo, sigue existiendo un "proceso escritor" vivo, de forma que el propio proceso padre se queda en espera a él mismo escriba en la tubería. El indicador de que no hay escritores activos – *eof* – nunca se envía.

Ej. 13) Shell

c)

i. Se ha empleado la función *execvp*, porque permite introducir los argumentos del comando a ejecutar en una cadena de caracteres. Esta forma resulta ser la mas conveniente, ya que evita tener que separar los argumentos entre sí. No se podría haber empleado una función *exec* de la familia *l*, ya que no se conoce el número de argumentos de entrada de antemano. Asimismo, se podrían haber empleado *execv*, con la tarea adicional de indicar el *path* del ejecutable.

ii. Al ejecutar el comando, en el fichero *log.txt* aparece la siguiente información:

```
Exited with value 127
```

Se trata del valor enviado por /bin/sh cuando la orden no se encuentra en el PATH.

iii. Tras ejecutar el programa que termina – de nombre abort.c y ejecutable *abort* - con *abort()*, se obtiene el siguiente registro en el fichero *log.txt*:

```
Terminated by signal 6
```