PRÁCTICA 1

SISTEMAS OPERATIVOS

Jorge de Miguel Pires
Laura de Paz Carbajo
Grupo 2202
18/02/2020

EJERCICIO 1

a) Comando: `man -k pthread`

Lista de funciones obtenidas al ejecutar el comando anterior:

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) Ejecutamos primero el comando `man man` para hallar la sección de llamadas al sistema o system calls, y vemos que es la 2. Ejecutamos, entonces, el comando `man 2 write` para obtener información sobre esta función.

EJERCICIO 2

a) Comando: `grep molino "don quijote.txt" >> aventuras.txt`
grep busca las líneas que contengan "molino"y en vez de imprimirlas por pantalla, con los caracteres ">>" las añade al final del archivo aventuras.txt, es decir, gracias a estos caracteres conseguimos redirigir la salida del comando al archivo de texto.

b) Comando: `ls | wc -l`
Con ls obtenemos la lista de ficheros del directorio actual y utilizo ese resultado como si fuera un archivo para contar el número de líneas.

c) Comando: `cat "lista de la compra Pepe.txt" "lista de la compra Elena.txt" | sort | uniq | wc -l > numcompra.txt`
Con `cat` se concatenan ambos ficheros, se ordena la salida de la concatenación con `sort` y se obtienen las líneas distintas con el comando `uniq`. A continuación, se utiliza la salida de `uniq` para contar el número de líneas mediante el comando `wc -l` y se guarda este número en el archivo numcompra.txt redirigiendo la salida del pipeline mediante **">"**.

EJERCICIO 3

a) Al intentar abrir un fichero que no existe se muestra el mensaje: "No such file or directory" que corresponde al valor "ENOENT 2" de errno.

b) Al intentar abrir ese fichero se obtiene "Permission denied" que corresponde al valor de errno "EACCES 13"

c) Habría que almacenar el valor de errno previamente en una variable de tipo int para asegurarse de conservar su valor si se llamase a otra función de por medio, ya que, aunque no ocurra ningún error en esta, se podría modificar el valor de errno.

EJERCICIO 4

a) Al utilizar la función `clock()` en un bucle para hacer esperar al programa 10 segundos, si ejecutamos en otra ventana de la terminal el comando `top` podemos observar que todos los

recursos del sistema (CPU) los utiliza el programa que acabamos de crear. Durante esos 10 segundos el 100% de la CPU está siendo utilizada por nuestro programa en ejecución.

b) Al crear el mismo programa pero con la función `sleep()`, este no consume ningún recurso, es decir, no aparece en la lista de procesos de `top`, ya que está esperando sin utilizar recursos de la CPU ya que el planificador pasa a ejecutar otros procesos mientras nuestro programa espera a que pase el tiempo especificado.

EJERCICIO 5

a) Al no esperar a los hilos (eliminando `pthread_join`) el proceso abaca antes de que lo hagan los hilos por lo que la función `slow_printf()` no se ejecuta completamente y, dependiendo de cuando se ejecute aparecen las letras “H”, “H M” o ninguna.

b) Al reemplazar `exit` por `pthread_exit` el programa no termina hasta que todos los hilos han acabado.

c) Para que sea correcto el no esperar a los hilos debemos añadir las dos siguientes líneas de código para “desligar” los hilos del proceso y que sus recursos se liberen automáticamente:

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

EJERCICIO 6

Archivo `ejercicio_hilos.c` .

EJERCICIO 7

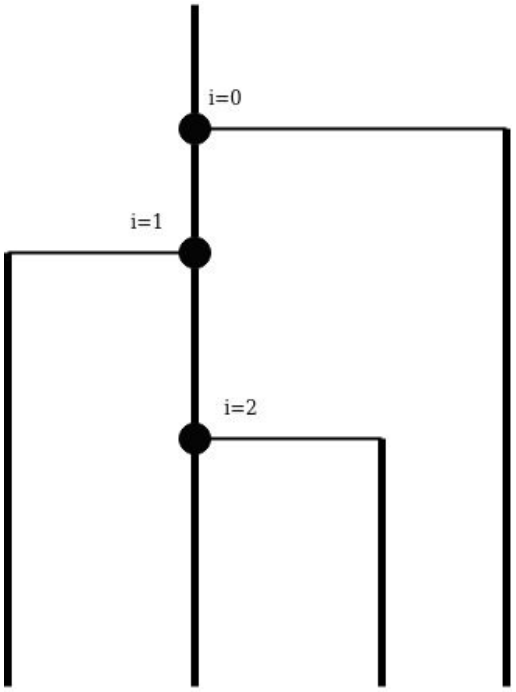
a) En principio, antes de ejecutar el programa no se puede saber en qué orden se imprimirá el texto. Sin embargo, al ser un programa tan rápido, la mayoría de las veces se ejecutará el proceso padre antes que los hijos.

b) Sustituimos la línea de código en la que se imprime el PID del hijo por:

```
printf("Hijo %d   PID = %jd   PPID = %jd\n", (intmax_t)getpid(),
(intmax_t)getppid());
```

Esta modificación imprime el PID del proceso hijo que se está ejecutando y el PPID (el identificador del proceso padre). Esto lo obtenemos con las funciones `getpid()` y `getppid()` que devuelven enteros de tipo `pid_t`, que necesitamos convertir a un integer muy grande, en este caso, `intmax_t` para poder imprimirlo mediante un casting.

c)



El árbol de procesos de este ejercicio tiene esta forma ya que en el bucle se llama a la función `fork()` y se crea un proceso hijo del proceso ejecutándose en ese momento. Cuando el proceso ejecutándose no es el proceso padre, cumple la condición `pid == 0`, ya que `fork()` devuelve 0 cuando es el proceso hijo el que se está ejecutando, y saldrá del programa por la instrucción `exit(EXIT_SUCCESS)`. Esto hace que los procesos hijos no llamen de nuevo a form y no creen sus propios procesos hijos.

d) Sí, dado que el `wait(NULL)` se encuentra fuera del bucle, por lo que únicamente se espera al hijo que acabe el primero y entonces el proceso padre acaba, por lo que los demás se quedan huérfanos y, cuando acaben, zombies.

e) Para solucionar esto únicamente hay que introducir el `wait(NULL)` dentro del bucle, lo que provoca que en cada iteración se espere al proceso hijo que se acaba de crear debido a la llamada de la función `fork()`.

EJERCICIO 8

Archivo ejercicio_arbol.c

EJERCICIO 9

a) Cuando se ejecuta el código únicamente aparece por pantalla el mensaje “Padre: ”. Este programa no es correcto ya que, al hacer la llamada a la función `fork()` y crear un proceso diferente, este nuevo proceso no comparte las mismas direcciones de memoria que el proceso padre, por lo que al modificar la variable **“**`sentence`**”** el nuevo valor de esta únicamente se almacena en el espacio de memoria reservado para el proceso hijo. La variable del proceso padre está inicializada pero no se ha copiado el mensaje en dicha variable.

b) Para solucionar la fuga de memoria habría que liberar el array en ambos procesos, padre e hijo, ya que al crear el proceso hijo se crea una copia del array en otra dirección de memoria distinta por lo que se crea un nuevo array. El código quedaría de la siguiente forma:

```
int main (void) {

  ...

  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) ejercicio_shell.c

b) Hemos decidido utilizar la función `excvp`, ya que es más cómodo para nosotros al insertar los comandos a ejecutar, dado que con esta función no hace falta especificar el origen de dicho comando. También se podría haber usado las funciones `execv` o `execve`, que funcionan de manera parecida, solo que hay que especificar dónde está contenido el comando a ejecutar.

c) Al ejecutar el comando `sh -c inexistente` se imprime:

```
sh: 1: inexistente: not found
Exited with value 127
```

d) Al crear un programa que termine en `abort()` y ejecutarlo en la shell aparece el siguiente mensaje:
`Terminated by signal 6.`

EJERCICIO 11

Para analizar los distintos apartados de este ejercicio hemos creado un programa test.c que contiene un bucle infinito.
Primero encontramos el PID de dicho proceso con el *pipeline* `ps -A | grep test` y en el directorio `/proc` accedemos al directorio cuyo nombre coincide con el PID de nuestro proceso.
Con `ls -l` encontramos los datos que se nos piden:

a) exe -> /home/alumnos/e399596/UnidadH/SOPER/Practica1/test
b) cwd -> /home/alumnos/e399596/UnidadH/SOPER/Practica1
c) Utilizamos `cat cmdline` y nos muestra ./test
d) Utilizamos el *pipeline* `cat environ | tr '\0' '\n' | grep LANG` y nos muestra LANG=es_ES.UTF-8
e) Si utilizamos el siguiente comando: `cat status | grep Threads` nos muestra que nuestro programa tiene 1 hilo (Threads = 1). Para ver los distintos hilos del proceso debemos meternos en el directorio `task` del proceso, donde salen todos los hilos que este tiene.

EJERCICIO 12

a) En el **Stop 1**, al inspeccionar los descriptores de ficheros que el proceso tiene abiertos, observamos que únicamente se encuentran abiertos los ficheros 0, 1 y 2 correspondientes a stdin, stdout y stderr respectivamente.

b) En el **Stop 2** se crea un nuevo fichero llamado `file1.txt` que obtiene el descriptor número 3 y en el **Stop 3** ocurre lo mismo, se crea un nuevo fichero llamado `file2.txt` al cual le corresponde el descriptor número 4.

c) Tras el **Stop 4**, al ver la lista de descriptores del proceso nos encontramos con que el fichero `file1.txt` ha sido eliminado. Sin embargo, no ha sido borrado de disco, simplemente se ha desvinculado del proceso y aún se puede acceder a su contenido usando la orden `cat 3` en el directorio de descriptores de fichero del proceso (con esto conseguimos conocer el contenido del fichero, el cual era “Hello”). Este comportamiento se debe a que el proceso simplemente se ha desvinculado del archivo pero aún existe alguna ruta de acceso a este fichero por lo que se conservará en disco hasta que todas estas rutas desaparezcan.

d) En el **Stop 5** se observa que el descriptor 3 ha desaparecido, esto se debe a que el proceso a cerrado definitivamente el fichero `file1.txt` y ya no se puede acceder a su contenido desde `/proc`. En el **Stop 6** se crea un nuevo fichero llamado `file3.txt` que obtiene el descriptor número 3, es decir, el antiguo descriptor del fichero `file1.txt` que

hemos cerrado anteriormente. Con este ejemplo llegamos a la conclusión de que los descriptores de ficheros, una vez que se cierra el fichero al que estaban ligados, se reasignan a otros nuevos ficheros que el proceso vaya abriendo. En el **Stop 7** el programa abre de nuevo el fichero `file3.txt` pero esta vez se hace con la flag `O_RDONLY` por lo que el S.O. le asigna un nuevo descriptor, en este caso el 5.

EJERCICIO 13

a) “Yo soy tu padre” se imprime 2 veces por pantalla, ya que stdout se abre como un objeto de tipo FILE y antes de imprimir por pantalla, se guarda en un buffer. Al hacer el `fork()`, el buffer todavía contiene dicha frase y lo que ocurre es que se imprime la frase “Nooooo” y seguidamente “Yo soy tu padre” de nuevo.

b) Al poner `\n` al final de los mensajes este problema ya no ocurre, ya que lee el `\n` e interpreta que es una línea nueva, por lo que vacía el buffer.

c) Al redirigir la salida a un fichero vuelve a ocurrir igual que en el apartado a) ya que el programa guarda todo en el buffer aunque vea un salto de línea y, finalmente, lo imprime en el fichero, por lo que duplica la frase “Yo soy tu padre”.

d) Para corregir definitivamente este problema sin dejar de usar el printf se deberá añadir la siguiente línea tras cada llamada a la función printf: `fflush(stdout);`

EJERCICIO 14

a) Al ejecutar el código, la salida que obtenemos es la siguiente:

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

b) Si el proceso padre no cierra el extremo de escritura el programa sigue ejecutándose ya que al existir un escritor, el proceso no reconoce que la lectura haya acabado y sigue intentando leer hasta llegar a un EOF, pero esto no ocurre ya que no se cierra el extremo escritor de la pipe.

c) Al realizar estos cambios, observamos que no se imprime nada por pantalla. Esto se debe a que, al finalizar el proceso padre, el hijo se queda huérfano y, al finalizar, pasa al estado de zombie, ya que ningún proceso recupera su información de retorno y, por lo tanto, no llega a imprimir el mensaje de “He escrito en el pipe”.

EJERCICIO 15

ejercicio_pipes.c