Junco de las Heras && Marta Vaquerizo

SISTEMAS OPERATIVOS

PRÁCTICA 1

Ejercicio 1: Uso del manual.

a) El comando usado en este apartado es:

```
man –k pthread > pthread.txt
```

donde pthread.txt es el fichero en el que está la lista de las funciones de manejo de hilos, incluido en la misma carpeta que este documento.

Resultado:

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) Al ejecutar el comando “man man”, se observa que las llamadas al sistema están en la sección 2.

Para buscar información sobre la llamada al sistema “write”, se usa el comando:

man 2 write

Ejercicio 2: Comandos y Redireccionamiento.

a) En este apartado, se usa una tubería para abarcar el problema:

cat "don quijote.txt" | grep molino >> aventuras.txt

Primero se imprime el contenido de “don quijote.txt” en stdout y con un pipeline se le da a la entrada a un grep, que buscará la palabra molino y todos los matchs los guardará en aventuras.txt

b) Para calcular el número de fichero que hay en el directorio actual, se ejecuta la siguiente tubería:

ls | wc | awk ‘{print $1}’

Primero, se ejecuta el comando “ls” para saber que hay en el directorio actual. Con esa información, se ejecuta “wc”, que cuenta las palabras, las letras y líneas de los ficheros del directorio, y saca esta información por pantalla, en ese orden. Y por último, para escoger el número de ficheros, se ejecuta el comando “awk ‘{print $1}’”, para obtener la primera columna, que es la que nos interesa.

c) La tubería pensada para resolver el problema planteado es:

cat 'lista de la compra Elena.txt' 'lista de la compra Pepe.txt' 2> /dev/null | sort | uniq | wc | awk '{print $1}' > "num compra.txt"

Primeramente, se concatenan los dos ficheros, y si alguno de los dos no existe, la salida de error (stderr o 2) se redirige a /dev/null. Luego de la concatenación, se ordenan las filas para hacerle el uniq (que solo elimina filas repetidas contiguas) y se hace un “uniq”. Se cuentan las palabras, las letras y líneas de los ficheros de la salida de “uniq” con “wc” y se elige la primera columna con “awk ‘{print $1}’” redirigiendo la salida al fichero “num compra.txt”.

d) El comando para contar el número de hilos de cada proceso del sistema y lo guarde en hilos.txt es:

ps -A -L | awk '{print $1}' | uniq –c > hilos.txt

Primero con ps –A, se obtienen todos los procesos del sistema, y con –L los hilos. Ahora para contar cuántos hilos tiene cada proceso se puede hacer de más maneras, pero nosotros usaremos un uniq -c a las primeras columnas devueltas con awk para que te cuente los procesos repetidos tantas veces como hilos tengan, y se redirige a hilos.txt.

EJERCICIO 3: Control de Errores.

Se crea un .c para testear (ejercicio3.c):

a) El mensaje que se imprime por pantalla si se intenta leer un fichero no existente es:

No such file or dictory

Es decir, que no encuentra el fichero. Este mensaje corresponde al valor 2 de errno.

b) Al intentar abrir el fichero /etc/shadow, el mensaje es:

Permission denied

Es decir, que no te deja leer ese fichero, y corresponde al valor 13 de errno.

c) Entonces será necesario guardarse errno en una variable auxiliar debido a que printf podría dar un fallo y alterar el valor de errno que nos interesa.

EJERCICIO 4: Espera Activa o Inactiva.

a) Se observa que en el top, durante esos 10 segundos, el proceso que lanza a.out está ejecutándose sin dejar libre el procesador y sin dejar que otro proceso se ejecute.

b) Sí, cambia que no aparece el nombre del proceso lanzado por a.out al ejecutar top. Eso significa que es marcado con estado de suspendido, a la espera de una señal de reloj, dejando el procesador libre para otros procesos.

EJERCICIO 5: Finalización de hilos.

a) Si no espera a los hilos creados, una vez el hilo principal acaba el programa este elimina a los hilos creados, no pudiendo ellos (probablemente) acabar correctamente la función que estaban haciendo.

b) Si se elimina exit por pthread_exit, el hilo principal se quedará esperando a que los dos hilos que ha lanzado acaben. Por contra, los recursos de los hilos no se liberan, ya que no hacen un join (Comprobado con Valgrind). Esto es debido a que un pthread por defecto es joinable, y necesita que algún otro hilo recoja con pthread_join su retorno antes de liberar sus recursos.

c) Si eliminamos pthread_join entonces tenemos que convertir esos pthread en detached, usando pthread_detach(pthread_self()) en slow_printf, en la función que ejecuta los hilos creados por el hilo principal. Seguimos teniendo la obligación de poner antes de que se haga exit el pthread_exit(), para que el hilo principal espere a que acaben los hilos que ha creado.

EJERCICIO 6: Creación de Hilos y Paso de Parámetros.

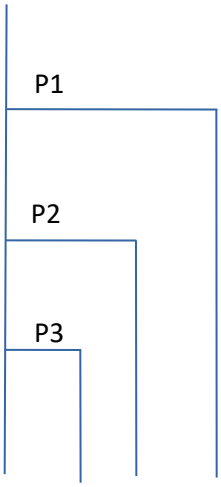
El código, que satisface los requisitos propuestos, está en ejercicio_hilos.c. Para más descripción lea los comentarios.

EJERCICIO 7: Creación de procesos.

a) No, no se puede saber *a priori* que proceso se va a ejecutar primero, ya que es el Sistema Operativo el que, según el planificador, ejecuta un proceso u otro, pudiendo algunas veces uno acabar antes que el otro y viceversa.

b) En el código se ha añadido una variable pid_t padre, y se ha guardado en esa variable, el PID del padre (el que se está ejecutando) antes del fork(). Se ha decidido llevar a cabo esta estrategia, ya que cuando un proceso padre lanza un proceso hijo, en éste último, se copia toda la información del proceso padre (stack, heap,…), y entre esa información están las variables locales, por lo que si antes de lanzar el proceso hijo, se guarda en la variable local el PID del padre, el hijo va a poder acceder a esa variable local, es decir, al PID del padre. Usamos la instrucción: printf ("Hijo %d %d\n", getpid(), ppadre);

c) En el siguiente esquema se muestra el árbol de procesos generado por el código del ejercicio.



El proceso con el que empezamos hace un fork(), sacando un hijo. Este hijo entra en su if y hace un exit(), pero el padre se mete de nuevo en el bucle y hace otro fork(), pasando lo mismo que antes. Esto se ejecuta NUM_PROC veces, en nuestro caso NUM_PROC = 3.

d) Sí, deja huérfanos, ya que el wait(NULL) termina y pasa a la siguiente instrucción si alguno de los procesos hijos ha acabado, esto no garantiza que todos los procesos hijos hayan acabado, ya que puede ser que muera 1 de los 3 que ha enviado y que los otros 2 todavía no hayan acabado.

e) La modificación que se ha hecho en el código para que no queden procesos huérfanos es poner tras acabar el for:

```
while (wait(NULL) != -1) {

}
```

De esta manera, hasta que wait(NULL) no sea -1 (en ese caso todos los procesos hijos han acabado), esperará.

EJERCICIO 8: Árbol de procesos.

El código, que cumple lo propuesto en el ejercicio, está en ejercicio_arbol.c. Para más información lea los comentarios.

EJERCICIO 9: Espacio de Memoria.

a) Lo que ocurre es que no se imprime por pantalla lo que se ha copiado en el proceso hijo. Esto se debe a que el padre no tiene esa información, ya que el proceso hijo y el proceso padre están en distintas zonas de la memoria.

Sí que es cierto que el hijo reserva memoria como el padre, ya que el hijo es una copia del padre, pero el padre no tiene lo que el proceso hijo copia durante su ejecución.

Se puede concluir que el programa no es correcto porque hay una fuga de memoria al no liberarse el char* sentence;

b) En la captura siguiente, se puede observar que se han incluido dos free's, uno para cada proceso, tanto para el padre como para el hijo. Esto se debe a que el hijo es una copia del padre, por lo que como el padre había reservado memoria, el hijo también. Entonces, cada uno libera su copia correspondiente. Se puede comprobar con Valgrind que no hay pérdidas de memoria al realizar este cambio.

```
pid = fork();
if (pid < 0)
{
    perror("fork");
    exit(EXIT_FAILURE);
}
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: Shell.

a) El programa que cumple los requisitos de este apartado se encuentra en la misma carpeta y se llama ejercicio_shell.c.

b) Se ha utilizado execvp, que recibe como argumentos el path del proceso que lo llama (lo recibe de forma implícita) y un vector con los argumentos, reservado de forma dinámica. Otra función que se podía haber usado es execv, ya que se pasa un array con los argumentos como en execvp, aunque, es un poco más tedioso que el execvp, que ya cada vez que se lanza un comando, se ha de pasar el path en el que se quiere ejecutar. Por otro lado, la función execl no se puede utilizar, ya que no se sabe cuántos argumentos se van pasar, debido a que depende de la instrucción que se ejecute.

c) Lo que aparece por pantalla cuando se ejecutas sh –c inexistente es lo siguiente:

sh: 1: inexistente: not found

Exit with value 127

d) El programa en el que se finaliza con abort se llama abort.c, y su ejecutable es abort.exe. Su output, cuando se ejecuta desde ejercicio_shell.c es:

Terminated by signal 6

e) El código está en ejercicio_shell_spawn.c. Es el mismo que el anterior pero hemos incluido spawn.h y hemos cambiado la llamada a fork+execvp por posix_spawnp(&pid, args[0], NULL, NULL, args, NULL).

EJERCICIO 11: Directorio de Información de Procesos

Se abre la terminal y se ejecuta: cd /proc/self. Desde allí se ejecutan los siguientes comandos y los resultados al ejecutarlos:

a) El comando para buscar el nombre del ejecutable es:

readlink exe

/bin/bash

b) El comando para buscar el directorio actual del proceso es:

readlink cwd (o también pwd, pero con self no te muestra el PID, sino self)

/proc/7267

c) La línea de comandos para lanzarlos fue:

cat cmdline | tr '\0' '\n'

bash

d) El comando para buscar el valor de la variable de retorno LANG es:

echo $LANG en US.UTG-8

e) El comando para buscar la lista de hilos del proceso y la lista:

ls task

7267

EJERCICIO 12: Visualización de descriptores de fichero.

a) Los descriptores de ficheros abiertos en el Stop 1 son los tres que al comenzar un programa ya están abiertos: el 0, 1 y 2. El 0 es stdin, el 1 es stdout y el 2 es stderr.

b) En el Stop2 se ha hecho open a FILE1 y se ha creado el descriptor de fichero 3 y en el Stop3 se ha hecho open a FILE2 y se ha creado el descriptor de fichero 4. La tabla de descriptores de fichero queda 0 1 2 3 4.

c) No se ha borrado el fichero todavía debido a que sigue abierto en el proceso que estamos ejecutando. Una vez que el proceso muera, o se haga close(file1) el fichero será borrado. Lo que ha pasado es que se ha borrado el symbolic link, por lo que desde un directorio no se puede acceder, tampoco desde proc/<pid>/fd/3, ya que dice que está borrado. Un método para recuperar los datos sería dumpear (volcar los datos) el fichero en otro, con el siguiente código de ejemplo pread(file1, &c, 1, i) lee el i-ésimo offset del descriptor de fichero y lo guarda en el char c, que es escrito a otro descriptor de fichero que teníamos que haber creado anteriormente.

```
char c;
  for(int i = 0; ; i++) {
     if(pread(file1, &c, 1, i) <= 0) {
        break;
     }
     printf(":%c:\n", c);
     if(write(file2, &c, 1) <= 0) {
        perror("write");
```

```
        exit(EXIT_FAILURE);
    }
  }
```

d) Antes del Stop 5 se ha cerrado el descriptor de fichero del file1, por lo que tanto el descriptor de fichero como el propio fichero se cierran y se borra, respectivamente. La tabla quedaría: 0 1 2 4. Antes del Stop 6 se crea un nuevo fichero (file3), por lo que habrá una nueva entrada en la tabla. Este nuevo descriptor corresponde al número 3, es decir, se reutiliza el descriptor del file1 ya que este ya no existe. Y antes del Stop 7, se crea un nuevo descriptor de fichero al hacer open, quedándose la tabla 0 1 2 3 4 5.

EJERCICIO 13: Problemas con el buffer.

a) El mensaje "Yo soy tu padre" sale dos veces por pantalla. Esto se debe a que como el proceso hijo es una copia del proceso padre, y la cadena de caracteres "Yo soy tu padre" está en el buffer, el hijo también tiene otro buffer, pero con esa misma información. Es por eso que, antes de acabar el proceso hijo, imprime "Yo soy tu padre" otra vez, imprimiéndose lo siguiente por pantalla: Yo soy tu padreNoooooYo soy tu padre.

b) No ocurre lo mismo, aparece una vez por pantalla, ya que al poner el \n en los mensajes, se fuerza a que saque por pantalla de mensaje en mensaje, vaciando el buffer cada vez, y es por eso que en buffer del proceso hijo no está "Yo soy tu padre".

c) Ocurre lo mismo que en el primer caso, porque al pasarlo a un fichero, el \n no funciona como en la salida estándar.

d) Para que no ocurra esto, habrá que poner la función fflush (stdout) tras cada mensaje, para que lo que haya en el buffer, tras cada mensaje, salga por pantalla, vaciándolo después.

EJERCICIO 14: Ejemplo de Tuberías

a) Al ejecutar el código aparece lo siguiente por pantalla:

He escrito en el pipe

He recibido el string: Hola a todos!

b) Al ejecutar el código sin que el padre cierre el extremo de escritura, sale lo siguiente por pantalla:

He recibido el string: Hola a todos!

He escrito en el pipe

y se queda esperando, sin terminar, ya que al no cerrar el padre el extremo de escritura, no se sabe si ha finalizado el proceso de escritura, y por lo tanto el hijo, quien escribe, no termina, y como el padre espera a que el hijo termine, no se acaba el programa.

c) Al esperar 1 segundo antes de escribir el padre que lee también espera 1 segundo, luego cuando el hijo escribe el padre lee y acaba correctamente como antes.

Al acabar sin leer y sin esperar no imprime nada por pantalla, ya que el padre no lee lo que escribió el hijo, pero el hijo tampoco escribe “he escrito en el pipe” porque hace un sleep de 1 segundo, lo que le da tiempo al padre a acabar su ejecución, y al morir, mata al hijo (esto es medido experimentalmente, podría darse el caso que el hijo acabase el sleep sin que el procesador le hubiese dado tiempo al padre a ejecutarse y entonces sí que escribiría por pantalla). Al ejecutarse con Valgrind recibimos SEGPIPE, debido a que acabamos un proceso sin haber acabado de leer el pipe.

EJERCICIO 15: Comunicación entre Tres Procesos.

El código que cumple los requisitos propuestos en el ejercicio está en ejercicio_pipes.c. Para más información lea los comentarios.