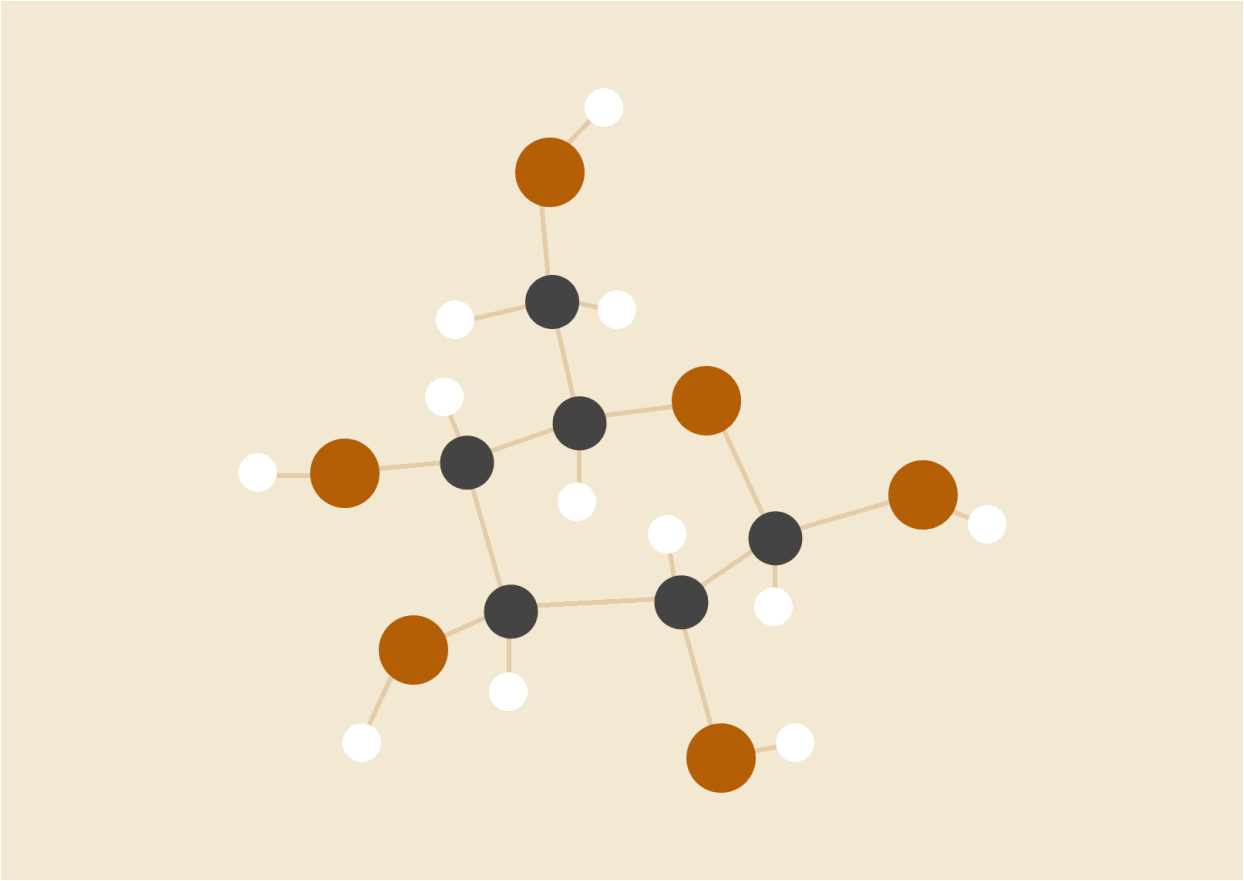
Sistemas Operativos

Práctica 1



Kevin de la Coba y José Manuel Freire

20/02/2020

INGENIERÍA INFORMÁTICA

INTRODUCCIÓN

En esta primera práctica, aprenderemos el funcionamiento de la shell, los procesos, hilos, la ejecución de programas, el uso de ficheros y por último el uso de tuberías.

SEMANA 1 - INTRODUCCIÓN A LA SHELL. HILOS.

Ejercicio 1: Uso del Manual.

a) Buscar en el manual la lista de funciones disponibles para el manejo de hilos y copiarla en la memoria junto con el comando usado para mostrarla. Las funciones de manejo de hilos comienzan por "pthread".

```
pthread_attr_destroy (3) - initialize and destroy thread attributes object
pthread_attr_getaffinity_np (3) - set/get CPU affinity attribute in thread attrib
utes object
pthread_attr_getdetachstate (3) - set/get detach state attribute in thread attrib
utes 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 threa
d attributes object
pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in thread a
ttributes object
pthread_attr_getscope (3) - set/get contention scope attribute in thread attribut
es object
pthread_attr_getstack (3) - set/get stack attributes in thread attributes object
pthread_attr_getstackaddr (3) - set/get stack address attribute in thread attribu
tes 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 attrib
utes object
pthread_attr_setdetachstate (3) - set/get detach state attribute in thread attrib
utes 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 threa
d attributes object
pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in thread a
ttributes object
pthread_attr_setscope (3) - set/get contention scope attribute in thread attribut
es object
pthread_attr_setstack (3) - set/get stack attributes in thread attributes object
pthread_attr_setstackaddr (3) - set/get stack address attribute in thread attribu
tes 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 ha
ndlers 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 han
dlers 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 mu
tex 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 mu
tex attributes object
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the threa
d read-write lock attribute object
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the threa
d 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_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 threadspthread_testcancel (3) - request delivery of
any pending cancellation request
```

```
man -k pthread > 1_a
```

Con el comando man -k y la string "pthread" mostramos todo lo relacionado con las funciones de manejo de hilos, una vez conseguido el output deseado podemos redirigir este con el símbolo '>' al archivo que queramos en nuestro caso "1_a".

b) Consultar en la ayuda en qué sección del manual se encuentran las “llamadas al sistema” y buscar información sobre la llamada al sistema write. Escribir en la memoria los comandos usados.

Al usar la función man man, podemos ver que las llamadas al sistema están en la sección 2 del manual. Usando man 2 write podemos obtener toda la información sobre la llamada al sistema write. En caso de solo poner man write, nos daría los resultados de otro write (Executable programs or shell commands).

Ejercicio 2: Comandos y Redireccionamiento.

a) Escribir un comando que busque las líneas que contengan “molino” en el fichero “don quijote.txt” y las añada al final del fichero “aventuras.txt”. Copiar el comando en la memoria, justificando las opciones utilizadas.

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

Con grep buscamos las palabras (en este caso "molino") que existan dentro de un archivo (en este caso "quijote.txt") y luego con el símbolo >> añadimos al final del archivo "aventuras.txt" las líneas donde aparece molino.

b) Elaborar un pipeline que cuente el número de ficheros en el directorio actual. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.

Usando el comando ls -l | wc -l conseguimos el número de archivos en un directorio. ls -l crea una lista con una línea por archivo y wc -l cuenta las líneas, obteniendo así el total de archivos.

c) Elaborar un pipeline que cuente el número de líneas distintas al concatenar “lista de la compra Pepe.txt” y “lista de la compra Elena.txt” y lo escriba en “num compra.txt”. Si alguno de los ficheros no existe, hay que ignorar los mensajes de error, para lo cual se redirigirá la salida de errores a /dev/null. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.

cat lista\ de\ la\ compra\ Elena.txt lista\ de\ la\ compra\ Pepe.txt | sort -u |wc -l > num\ \compra.txt

Primero concatenamos los dos archivos con el comando cat, después hacemos un

pipeline con sort -u, con -u cogemos las líneas que son únicas, volvemos ha hacer un pipeline con wc -l contamos el número de líneas, palabras y letras, al hacer -l nos muestra unicamente el número de líneas y para poner toda la información obtenidas en un archivo en concreto usamos el símbolo ">" y así lo introducimos en el archivo num\ \compra.txt

Ejercicio 3: Control de Errores.

a) ¿Qué mensaje se imprime al intentar abrir un fichero inexistente? ¿A qué valor de errno corresponde?

"No such file or directory", el valor de errno es = 2.

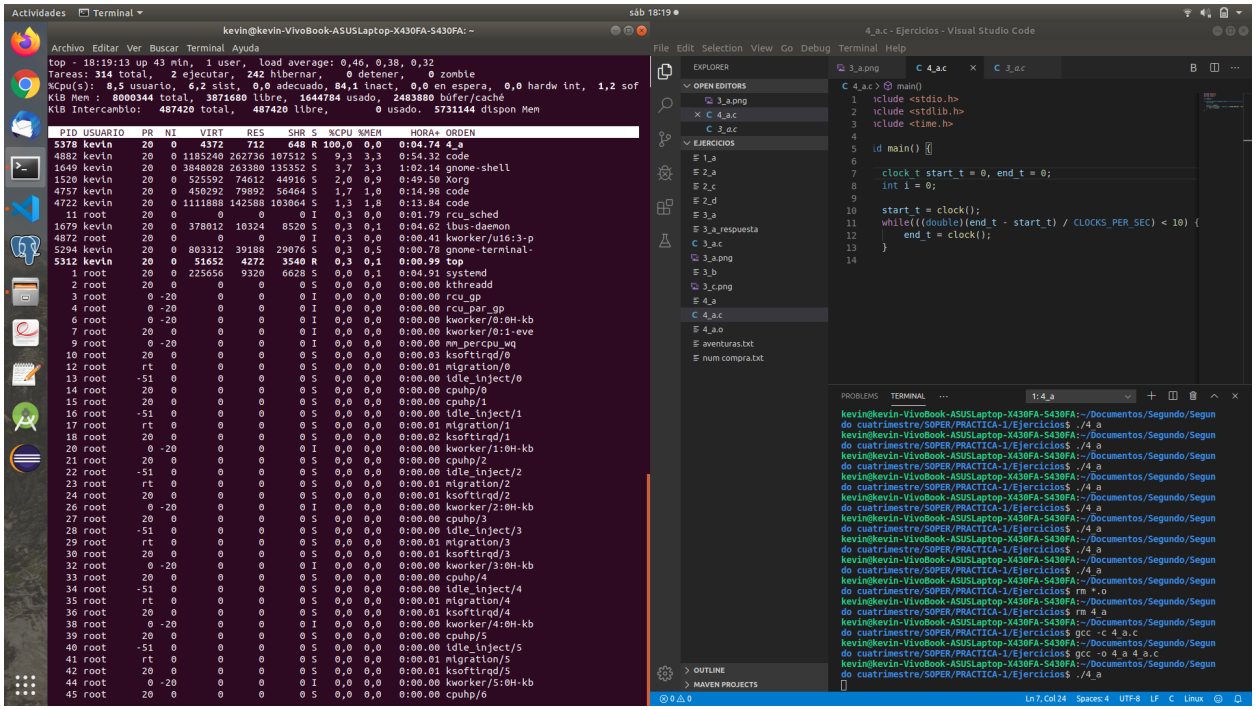
b) ¿Qué mensaje se imprime al intentar abrir el fichero /etc/shadow? ¿A qué valor de errno corresponde?

"Permission denied" valor de errno = 13

Ejercicio 4: Espera Activa e Inactiva.

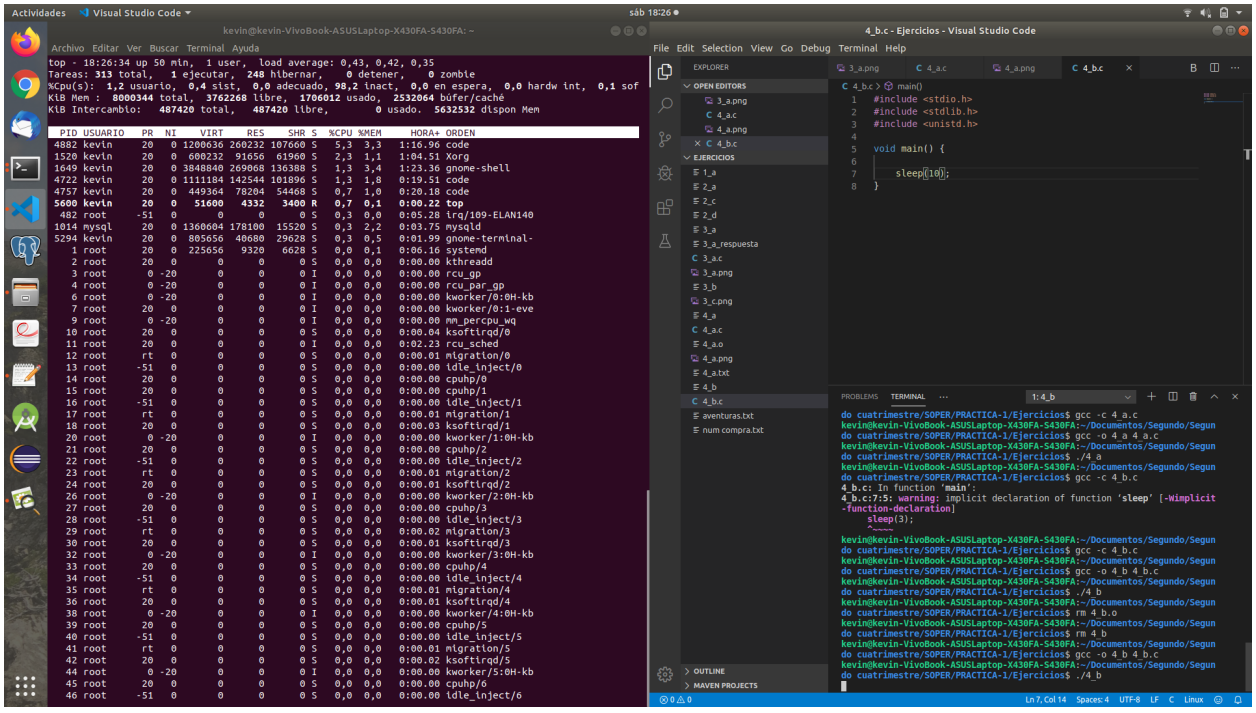
a) Escribir un programa que realice una espera de 10 segundos usando la función clock en un bucle. Ejecutar en otra terminal el comando top. ¿Qué se observa?

Después de escribir el programa, compilarlo y ejecutarlo, lo que se puede observar al hacer el comando "top" es como existe un proceso que ocupa el 100% de la cpu, ese proceso tiene como "ORDEN" el nombre del ejecutable que se está ejecutando.



b) Reescribir el programa usando sleep y volver a ejecutar top. ¿Ha cambiado algo?

Sí, lo que ha cambiado es que al ejecutar el comando top no podemos ver el proceso del programa, por el bajo uso de la cpu (en nuestro caso 4_b).



Ejercicio 5: Finalización de Hilos.

a) ¿Qué hubiera pasado si el proceso no hubiera esperado a los hilos? Para probarlo basta eliminar las llamadas a pthread_join.

b) Con el código modificado del apartado anterior, indicar qué ocurre si se reemplaza la función exit por una llamada a pthread_exit.

Al cambiar la función lo que ocurre es que, aunque el hilo principal termine, los otros dos siguen hasta que terminan de imprimir sus mensajes.

Ejercicio 6: Creación de Hilos y Paso de Parámetros.

a) Escribir un programa en C ("ejercicio_hilos.c") que satisfaga los siguientes requisitos:

Creará tantos hilos como se le indique por parámetro.

Cada hilo esperar 'a un número aleatorio de segundos entre 0 y 10 inclusive, que será generado por el hilo principal.

Después realizará el cálculo x 3 , donde x será el número del hilo creado. Por ´ultimo devolverá el resultado del cálculo en un nuevo entero, reservado dinámicamente.

El hilo principal deberá esperar a que todos los hilos terminen e imprimir todos los resultados devueltos por los hilos. Como la función pthread_create solo admite el paso de un único parámetro habrá que crear un struct con ambos parámetros (tiempo de espera y valor de x).

El programa deberá finalizar correctamente liberando todos los recursos utilizados.

Deberá asimismo controlar errores, y terminar imprimiendo el mensaje de error correspondiente si se produce alguno.

```
1  #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5 #include <sys/types.h>
 6 #include <sys/wait.h>
 7
 8 #define MAX_THREADS 100
 9
10 typedef struct{
```

```c
    int random;
    int x;
}Args;

void * xcube(void *voidp){
    int *resultado = NULL;

    resultado = (int*)malloc(sizeof(int));
    if (resultado == NULL) {

        perror("xcube");
        return NULL;
    }
    Args* args;
    args = (Args*) voidp;
    sleep(args->random);

    return (void*)resultado;
}

int main(int argc, char **argv){
    int hilos, error, i;
    pthread_t* threads=NULL;
    Args **args=NULL;
    int *voidp;


    if(argc<2){
        printf("Usar: ./a.out <número de hilos>\n");
        return 1;
    }

    hilos = atoi(argv[1]);
    if(hilos>MAX_THREADS) hilos = MAX_THREADS;

    threads = (pthread_t*) malloc (hilos*sizeof(pthread_t));
    if(!threads){
        perror("malloc");
        printf("Ha habido un error al alojar la memoria de threads.\n");
        return(EXIT_FAILURE);
    }
    args = (Args**) malloc (hilos*sizeof(Args*));
    if(!args){
        perror("malloc");
        printf("Ha habido un error al alojar la memoria args.\n");
        return(EXIT_FAILURE);
    }

    for(i=0;i<hilos;i++){
        args[i] = (Args*) malloc (sizeof(Args));
        args[i]->random = rand()%11;
        args[i]->x = i;
        error = pthread_create((threads[i]), NULL, xcube, args[i]);
        if(error!=0){
            perror("pthread_create");
            printf("Ha habido un error al crear el thread.\n");
            return 1;
        }
        error = pthread_join(threads[i], (void**)voidp);
        if(error!=0){
            perror("pthread_join");
            printf("Ha habido un error al hacer join en el thread.\n");
            return 1;
```

```
74          }
75          printf("El resultado es: %d\n", *voidp);
76      }
77
78      return 0;
79  }
```

Ejercicio 7: Creación de procesos.

a) Analizar el texto que imprime el programa. ¿Se puede saber a priori en qué orden se imprimirá el texto? ¿Por qué?

No, no se puede saber a priori debido a que los procesos creados se ejecutan a la vez, al ejecutarse a la vez puede pintear primero el hijo o el padre.

b) Cambiar el código para que el proceso hijo imprima su PID y el de su padre en vez de la variable i. Copiar las modificaciones en la memoria y explicarlas.

Al cambiar los prints podemos ver que siempre se imprime el mismo PID para el padre, pero el del hijo nunca es el mismo, esto es porque se generan procesos diferentes. Lo que también se puede ver es que sigue sin haber un orden concreto, ya que si lo ejecutamos varias veces el orden en que se printean los datos es diferente.

c) Analizar el árbol de procesos que genera el código de arriba. Mostrarlo en la memoria como un diagrama de árbol (como el que aparece en el Ejercicio 8) explicando por qué es así.

El final de un proceso puede cruzarse con el inicio de otro.

d) El código anterior deja procesos huérfanos, ¿por qué?

Porque el padre acaba antes que el último hijo formado, por lo que el hijo se queda como zombie, ya que es posible que se el padre cree otro hijo sin que el hijo anterior haya finalizado.

e) Introducir el mínimo número de cambios en el código para que no deje procesos huérfanos. Copiar las modificaciones en la memoria y explicarlas.

Para corregir el error de los huérfanos podríamos hacer un loop con la función "wait" con el fin de esperar a que todos los hijos terminen de ejecutarse.

while(wait(NULL) != -1) Por eso hacemos un loop que ejecute la función wait, hasta que devuelva error (-1) lo que significa que el proceso padre no tiene hijos.

Ejercicio 8: Árbol de procesos.

Escribir un programa en C (“ejercicio_arbol.c”) que genere el siguiente árbol de procesos: El proceso padre genera un proceso hijo, que a su vez generará otro hijo, y así hasta llegar a NUM_PROC procesos en total. El programa debe garantizar que cada padre espera a que termine su hijo, y no quede ningún proceso huérfano.

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NUM_PROC 3

int main(void) {

    /* Iniciamos la variable a cero para poder hacer un fork
*/
    pid_t pid = 0, *waitVar;

    for (int i = 0; i < NUM_PROC - 1; i++) {

```

```
18          /* Si eres el hijo, haces un fork */
19          if(pid == 0) {
20              pid = fork();
21          }
22          else if (pid < 0) {
23              perror("fork");
24              exit(EXIT_FAILURE);
25          } else {
26              /* El padre espera por todos sus hijos */
27              while(wait(NULL) != -1);
28          }
29      }
30
31      return(EXIT_SUCCESS);
    }
```

Ejercicio 9: Espacio de Memoria.

a) En el programa anterior se reserva memoria en el proceso padre y se inicializa en el proceso hijo usando la función strcpy (que copia un string a una posición de memoria). Una vez el proceso hijo termina, el padre lo imprime por pantalla. ¿Qué ocurre cuando se ejecuta el código? ¿Es este programa correcto? ¿Por qué?

Al ejecutar el programa, no se imprime el mensaje. Si la intención del programa es la de imprimir el mensaje, el programa no es correcto. Al hacer fork, se copia la BCP (bloque de control de procesos) del proceso, en ese momento en la memoria reservada no hay ninguna string en ninuguno de los dos procesos (me refiero al proceso padre y al proceso hijo). Con el "else if (pid == 0)" estamos haciendo que sea el hijo aquel que guarde en su variable la oración, y esta variable esta guardada en la BCP del proceso hijo la cual no tiene ninguna conexión con el proceso padre, al no haber conexión el proceso padre hace print de lo que tiene, nada.

b) El programa anterior contiene una fuga de memoria ya que el array sentence nunca se libera. Corregir el código para eliminar esta fuga y copiar las modificaciones en la memoria. ¿Dónde hay que liberar la memoria, en el proceso padre, en el hijo o en ambos? ¿Por qué?

La memoria se libera en el proceso padre, porque cuando un hijo muere, muere liberando todos los recursos que este ha usado, por lo que se podría eliminar memoria en los procesos que sean hijos pero seria innecesario, por lo que solo es necesario liberar recursos en el proceso padre.

Al hacer valgrind nos dice que hay memoria "still reachable" pero no hay errores.

Ejercicio 10: Shell.

a) Escribir el programa, satisfaciendo los siguientes requisitos: ...

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <wordexp.h>
#include <errno.h>

#define MAX 100
#define MAX_ARG 100

int main() {

    char string[MAX];
    char *arg = NULL;
    char *args[MAX_ARG];
    int i = 0, n = 0, err_v = 0, flag;
    pid_t pid;

    printf(">");

    /* Cogiendo el comando */
    while (fgets(string, MAX, stdin) != NULL) {

        /* Separando los argumentos */
        arg = strtok(string, " ");
        args[0] = (char*)malloc(strlen(arg)*sizeof(char));
        if (args[0] == NULL) return -1;
        strcpy(args[0], arg);
        i = 1;
        while (arg != NULL) {

            arg = strtok(NULL, " ");
            if (arg == NULL) {

                args[i] = NULL;
                break;
            }
            args[i] = (char*)malloc(strlen(arg)*sizeof(char));
            if (args[i] == NULL) return -1;
            strcpy(args[i], arg);
            i++;
        }

        if (i == 1) {

            for (n = 0; args[0][n] != '\n'; n++);
            args[0][n] = '\0';
        }

        /* Creando proceso */
        pid = fork();

        if (pid == 0) {
            execv(args[0], (char* const*)args);
        } else {
            wait(&flag);
        }

        if (WIFSIGNALED(flag)) {

            fprintf(stderr, "Terminated by signal %d\n",
WTERMSIG(flag));
        } else if (WIFEXITED(flag)) {
            perror("\nResult");
```

```
67              fprintf(stderr, "Exited with value %d\n", flag);
68          }
69
70          printf(">");
71
72          /* Liberando memoria */
73          for(int n = 0; n < i; n++) {
74              free(args[n]);
75              args[n] = NULL;
76          }
77      }
78
79      return -1;
    }
```

b) Explicar qué función de la familia exec se ha usado y por qué. ¿Podría haberse usado otra? ¿Por qué?

La función que hemos usado ha sido la “execv”. Hemos usado esta función ya que nos permite meter un numero indefinido de argumentos en el comando, ya que el segundo argumento de esta es un array de strings.

Ejercicio 11: Directorio de Información de Procesos.

a) El nombre del ejecutable.

cat & readlink /proc/$PID/exe

b) El directorio actual del proceso.

cat & readlink /proc/$PID/cwd

c) La línea de comandos que se usó para lanzarlo.

cat /proc/$PID/cmdline

d) **El valor de la variable de entorno LANG.**

cat /proc/$PID/environ | tr '\0' '\n'

e) **La lista de hilos del proceso.**

1. cd proc/$PID/task
2. ls

Ejercicio 12: Visualización de Descriptores de Fichero.

a) Stop 1. Inspeccionar los descriptores de fichero del proceso. ¿Qué descriptores de fichero se encuentran abiertos? ¿A qué tipo de fichero apuntan?

Se encuentran abiertos los descriptores 0, 1 y 2.

b) Stop 2 y Stop 3. ¿Qué cambios se han producido en la tabla de descriptores de fichero?

stop 2. Se ha añadido un descriptor de fichero más. el 3.

stop 3. Se ha añadido un descriptor de fichero más. el 4.

c) **Stop 4. ¿Se ha borrado de disco el fichero FILE1? ¿Por qué? ¿Se sigue pudiendo acceder al fichero a través del directorio /proc? ¿Hay, por tanto, alguna forma sencilla de recuperar los datos?**
Sí, se ha borrado el fichero ya que el proceso que lo abrió era el único que tenía una ruta abierta a él, por lo que al hacer unlink del fichero este se ha borrado del disco duro. No se sigue pudiendo acceder a él desde el directorio proc, lo único que se puede ver es como si hacemos cat en el descriptor de fichero que se creó al crear el fichero, nos pone "...file1.txt (deleted)". No hay forma de recuperar los datos.

d) **Stop 5, Stop 6 y Stop 7. ¿Qué cambios se han producido en la tabla de descriptores de fichero? ¿Qué se puede deducir sobre la numeración de un descriptor de fichero obtenido tras una llamada a open?**
stop 5 Ha desaparecido el descriptor de fichero relacionado el fichero "file1.txt".
stop 6 Se ha vuelto a crear el descriptor de fichero 3 pero el archivo que se ha creado es el file3.txt.
stop 7 Se ha creado otro descriptor de fichero, el 5.
La numeración del descriptor de fichero va en orden ascendente y si nos encontramos en la situación en la que borramos el descriptor de fichero N, pero existen descriptores de ficheros
mayores que N, cuando se cree un nuevo archivo este tendrá como descriptor de fichero el número N.

Ejercicio 13: Problemas con el Buffer.

a) **¿Cuántas veces se escribe el mensaje “Yo soy tu padre” por pantalla? ¿Por qué?**

Se imprime 2 veces. Al usar la función printf no estamos escribiendo en un buffer estamos escribiendo en el descriptor de fichero directamente pero como no está el carácter '\n' el hijo escribe todo lo que ha sido escrito.

b) **En el programa falta el terminador de línea (\n) en los mensajes. Corregir este problema. ¿Sigue ocurriendo lo mismo? ¿Por qué?**

El mensaje se escribe sin problema alguno. Porque cuando el hijo va a escribir este se encuentra en el descriptor de fichero un \n lo que le dice que no debe escribir nada que se encuentre ahí.

c) **Ejecutar el programa redirigiendo la salida a un fichero. ¿Qué ocurre ahora? ¿Por qué?**

El programa vuelve a escribir dos veces el mensaje del padre. Porque al usar una función que lee un objeto FILE, esta función usa un buffer, y al usar un buffer el padre escribe su mensaje en este y el hijo hace lo mismo, entonces cuando el hijo va a escribir en el descriptor de fichero este escribe todo lo que hay en el buffer.

d) Indicar en la memoria cómo se puede corregir definitivamente este problema sin dejar de usar printf.

setvbuf(pf, buffer, _IONBF, 16);

Con esta línea de código configuramos el use del buffer y poniendo el argumento _IONBF le decimos al sistema operativo que no use buffer.

Ejercicio 14: Ejemplo de Tubería.

a) Ejecutar el código. ¿Qué se imprime por pantalla?

He escrito en el pipe

He recibido el string: Hola a todos!

b) ¿Qué ocurre si el proceso padre no cierra el extremo de escritura? ¿Por qué?

Al parecer se queda esperando una respuesta y el programa no acaba. Como el proceso padre no ha cerrado su extremo no puede saber cuándo el proceso hijo ha acabado de escribir por eso el programa no acaba, porque en la línea de código nbytes = read(fd[0], readbuffer, sizeof(readbuffer)); primero devuelve 15 (que es lo que mide la string) y después devuelve 0, solo si se cierra el extremo de escritura, si no se cierra solo devuelve 15 y el padre se queda esperando a que este devuelva 0.

Ejercicio 15 Comunicación entre Tres Procesos.

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

int main(int argc, char **argv) {

    pid_t pid;
    int tuberia1[2], tuberia2[2];
    int numero;
    int pipe_status;

    srand(time(NULL));

    FILE *pf = NULL;
    pf = fopen("numero_leido.txt", "w");
    if (pf == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    /* Creando tuberias */
    pipe_status = pipe(tuberia1);
           if(pipe_status == -1)
           {
                      perror("pipe");
                      exit(EXIT_FAILURE);
           }

    /* Creamos dos procesos hijos */
    pid = fork();
    if (pid < 0){
                      perror("fork");
                      exit(EXIT_FAILURE);
           } else if (pid == 0) {

        /* Cierre del descriptor de entrada en el hijo */
                      close(tuberia1[0]);

        /* Generando número aleatorio */
        int random = rand()%11;

        ssize_t nbytes = write(tuberia1[1], &random, sizeof(int));
                     if(nbytes == -1)
                     {
                                perror("write");
                                exit(EXIT_FAILURE);
                     }

        /* Imprimiendo en pantalla el numero generado */
        printf("%d", random);

        exit(EXIT_SUCCESS);
    } else {

        /* Cierre del descriptor de salida en el padre */
                      close(tuberia1[1]);

                      /* Leer algo de la tubería... el saludo! */
                      ssize_t nbytes = 0;
                      do {
                                nbytes = read(tuberia1[0], &numero, sizeof(int));
                                if(nbytes == -1)
                                {
                                           perror("read");
                                           exit(EXIT_FAILURE);
                                }
                      } while(nbytes != 0);

                      while(wait(NULL) != -1);
    }

    pipe_status = pipe(tuberia2);
           if(pipe_status == -1)
           {
                      perror("pipe");
```

```
80                          exit(EXIT_FAILURE);
81                  }
82
83      pid = fork();
84      if (pid < 0){
85                          perror("fork");
86                          exit(EXIT_FAILURE);
87                  } else if (pid == 0) {
88
89              /* Cierre del descriptor de salida en el hijo */
90                          close(tuberia2[1]);
91
92                          /* Leer el numero en la tuberia */
93                          ssize_t nbytes = 0;
94              int numero_recibido;
95
96              nbytes = read(tuberia2[0], &numero_recibido, sizeof(int));
97              if(nbytes == -1)
98              {
99                  perror("read");
100                 exit(EXIT_FAILURE);
101             }
102
103             fprintf(pf ,"%d", numero_recibido);
104                         exit(EXIT_SUCCESS);
105     } else {
106
107             /* Cierre del descriptor de entrada en el padre */
108                         close(tuberia2[0]);
109
110             ssize_t nbytes = write(tuberia2[1], &numero, sizeof(int));
111                         if(nbytes == -1)
112                         {
113                                     perror("write");
114                                     exit(EXIT_FAILURE);
115                         }
116
117             wait(NULL);
118             fclose(pf);
119             exit(EXIT_SUCCESS);
120     }
121 }
```

CONCLUSION

Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto Escribe aquí tu texto.