INFORME PRÁCTICA 1

Alejandro Santorum Varela - alejandro.santorum@estudiante.uam.es
David Cabornero Pascual - david.cabornero@estudiante.uam.es
Pareja 7
Universidad Autónoma de Madrid

PENDIENTE

Contents

1 Introducción 2

2 Ejercicio 4-A 2

3 Ejercicio 4-B 3

4 Ejercicio 5-A 6

5 Ejercicio 5-B 7

6 Ejercicio 6 9

7 Ejercicio 8 11

8 Ejercicio 9 17

9 Ejercicio 12-A 23

10 Ejercicio 12-B 26

11 Ejercicio 13 30

12 Conclusión y comentarios finales 36

1 Introducción

Ese documento recoge los ejercicios realizados en la primera práctica de Sistemas Operativos.

Para cada ejercicio se comentará su diseño, su funcionalidad detalla y un análisis de las diferentes decisiones tomadas a la hora de enfrentarse a los problemas o dificultades encontradas. Además, se aportará el código en este informe para la comodidad del corrector, pero también se puede acceder al mismo en la carpeta de la entrega (donde está comentado), ignorando el código expuesto en este documento.

2 Ejercicio 4-A

Empezamos con el primero ejercicio entregable de la práctica, el ejercicio 4a. Se pedía examinar el código aportado y modificarlo tal que cada hijo imprima su PID y el PID de su proceso padre.

Se muestra a continuación el código modificado.

Código ejercicio 4a

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6
7  #define NUM_PROC 6
8
9  int main (void){
10   int pid;
11   int i;
12   int praiz;
13
14   praiz = getpid();
15
16   for (i=0; i <= NUM_PROC; i++){
17      if (i % 2 == 0) {
18         if ((pid=fork()) <0 ){
19            printf("Error al emplear fork\n");
20            exit(EXIT_FAILURE);
21         }
22         else if(getppid()==1 && getpid()!=praiz){
23            printf("Proceso huerfano %d\n", getpid());
24         }
25         else if(pid == 0){
```

```
26              printf ("PADRE %d\n", getppid());
27              printf("HIJO  %d\n", getpid());
28            }
29          }
30    }
31    exit(EXIT_SUCCESS);
32  }
```

Se ha modificado el código tal que ahora el hijo es el que imprime los PID's en lugar de que cada uno imprima el suyo. También se ha introducido una nueva comprobación (línea 22) para saber si el proceso ha quedado huérfano, cosa que se especifica al final del enunciado del ejercicio 4b.

Se muestra ahora la salida de este programa en Linux:

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio4a
PADRE 505
HIJO  506
PADRE 505
HIJO  507
PADRE 506
PADRE 505
HIJO  508
PADRE 507
HIJO  509
Proceso huerfano 506
Proceso huerfano 512
PADRE 506
Proceso huerfano 507
HIJO  510
PADRE 507
PADRE 508
Proceso huerfano 509
Proceso huerfano 516
Proceso huerfano 506
PADRE 509
PADRE 506
HIJO  511
HIJO  513
HIJO  514
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ HIJO  515
HIJO  517
Proceso huerfano 510
PADRE 510
Proceso huerfano 511
Proceso huerfano 514
PADRE 511
PADRE 514
HIJO  518
HIJO  519
HIJO  520
```

Se comentará esta salida junto con la salida del ejercicio 4b más adelante.

3 Ejercicio 4-B

El ejercicio 4b es una extensión del 4a, con la única modificación de incluir un wait(); al final del programa.

Código ejercicio 4b

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

#define NUM_PROC 6

int main (void){
  int pid;
  int i;
  int praiz;

  praiz = getpid();

  for (i=0; i <= NUM_PROC; i++){
     if (i % 2 == 0) {
        if ((pid=fork()) <0 ){
           printf("Error al emplear fork\n");
           exit(EXIT_FAILURE);
        }
        if(getppid()==1 && getpid()!=praiz){
           printf("Proceso huerfano %d\n", getpid());
        }
        else if(pid == 0){
           printf ("PADRE %d\n", getppid());
           printf("HIJO  %d\n", getpid());
        }
     }
  }

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

Igual que en el ejercicio 4a, se pedía que cada hijo imprimise su PID y el de su proceso padre, de ahí que los cambios sean idénticos. También se preguntaba por la funcionalidad de wait(), lo cual se comentará después de observar la salida de este programa.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio4b
PADRE 556
HIJO  557
PADRE 556
HIJO  558
PADRE 557
PADRE 556
PADRE 558
HIJO  559
PADRE 557
HIJO  560
HIJO  561
PADRE 556
PADRE 558
HIJO  562
PADRE 557
HIJO  563
HIJO  564
PADRE 559
PADRE 560
PADRE 561
HIJO  565
PADRE 559
PADRE 562
HIJO  566
HIJO  567
HIJO  568
HIJO  569
HIJO  570
Proceso huerfano 566
PADRE 566
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ HIJO  571
```

En este momento ya tenemos ejemplos suficientes, con la salida del 4a y del 4b, para comentar sus diferencias y el objetivo del wait().

Por un lado, lo primero que salta a los ojos es la diferencia de procesos huérfanos que genera. Esto es debido a que la función wait() en un proceso lo obliga a esperar a que **alguno** de sus procesos hijos (si los tiene) acabe su ejecución, por lo que la cantidad de procesos huérfanos es reducida, ya que es más complicado que algún proceso acabe después que su padre porque este tiene pocos hijos (en este programa) y espera por alguno.

Por otro lado, también se puede observar que el prompt de la terminal se realiza prácticamente después de finalizar la ejecución de todos los procesos (menos uno en este caso), a diferencia que el ejercicio 4a que saltaba cuando aún no habían finalizado aproximadamente un cuarto de los procesos.

Por último, se recomienda utilizar el comando pstree -p para percibir el árbol de procesos que se origina. Aquí un ejemplo.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum$ pstree -p
init(1)─┬─bash(2)───pstree(204)
        └─bash(34)───ejercicio4a(182)─┬─ejercicio4a(183)─┬─ejercicio4a(186)─┬─ejercicio4a(193)───ejercicio4a(197)
                                      │                  │                  └─ejercicio4a(194)
                                      │                  ├─ejercicio4a(189)───ejercicio4a(196)
                                      │                  └─ejercicio4a(191)
                                      ├─ejercicio4a(184)─┬─ejercicio4a(188)───ejercicio4a(195)
                                      │                  └─ejercicio4a(190)
                                      ├─ejercicio4a(185)───ejercicio4a(192)
                                      └─ejercicio4a(187)
```

Desafortunadamente no se puede obtener una buena panorámica de la diferencia de construcción-desaparición de procesos en el árbol entre los ejercicios 4a y 4b. De todas formas,

podemos predicir como va evolucionando. El proceso de construcción del árbol es idéntico en ambos casos (hasta un resultado final como el de la imagen); es en el transcurso de desaparición de nodos en el árbol en lo que varian los programas: en el primero, si pudiésemos ejecutar pstree cada vez que acaba un proceso, podríamos ver como muchos acaban sin proceso padre y son "adoptados" por el proceso de PID=1, init. En el segundo, podríamos ver que los procesos van desapareciendo prácticamente en orden, los padres después de alguno de sus hijos.

4 Ejercicio 5-A

En el ejercicio 5a se nos pide realizar los mínimos cambios posibles en el programa del ejercicio 4b de forma que se generen un conjunto de procesos hijos secuencial de modo que cada proceso tiene un único proceso hijo y ha de esperar a que finalice. Se muestra el código adoptado a continuación:

Código ejercicio 5a

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6
7  #define NUM_PROC 6
8
9  int main (void){
10   int pid;
11   int i;
12
13   for (i=0; i <= NUM_PROC; i++){
14      if (i % 2 != 0) {
15         if ((pid=fork()) <0 ){
16            printf("Error haciendo fork\n");
17            exit(EXIT_FAILURE);
18         }
19         else if(pid == 0){
20            printf ("PADRE %d \n",getppid());
21            printf("HIJO  %d \n",getpid());
22         }
23         else break;
24      }
25   }
26
27   wait(NULL);
```

```
28    exit(EXIT_SUCCESS);
29 }
```

El programa se ha modificado únicamente en la línea 23(sin contar el cambio de i%2==0 por i%2!=0 que especifica el enunciado), la sentencia `else break;` realiza la importante misión de que si el proceso no tiene la variable pid==0 quiere decir que está corriendo como padre, y por lo tanto no puede hacer más forks. Una forma muy buena de ver como se van originando los procesos es utilizar la función pstree -p enseñada en el ejercicio anterior.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum$ pstree -p
init(1)─┬─bash(2)───pstree(626)
        └─bash(34)───ejercicio5a(622)───ejercicio5a(623)───ejercicio5a(624)───ejercicio5a(625)
```

No hay duda de que cada proceso crea uno hijo y solo uno.

Además, se establecía que cada proceso tenía que esperar por la finalización de su hijo para acabar, lo cual es percibe bastante bien en la salida del programa.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio5a
PADRE 632
HIJO  633
PADRE 633
HIJO  634
PADRE 634
HIJO  635
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
```

A diferencia que en los ejercicios 4a y 4b, los PID's se imprimen en la terminal por orden, siempre en parejas PADRE-HIJO. En el ejercicio anterior esto no pasaba porque todos los programas estaban ejecutándose al mismo tiempo, haciendo que mientras uno imprimía PADRE-HIJO, otro proceso hacía lo mismo, dando un resultado en la terminal de, por ejemplo, PADRE PADRE HIJO HIJO.

5 Ejercicio 5-B

En el ejercicio 5b, como en el 5a, se nos pide realizar los mínimos cambios posibles en el programa del ejercicio 4b, pero ahora un único proceso padre origina un conjunto de procesos hijos y este ha de esperar a que acaben todos los procesos hijos antes de finalizar. Se muestra el código tomado a continuación:

Código ejercicio 5b

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
```

```
7  #define NUM_PROC 6
8
9  int main (void){
10   int pid;
11   int i;
12
13   for (i=0; i <= NUM_PROC; i++){
14      if (i % 2 != 0) {
15         if ((pid=fork()) <0 ){
16            printf("Error haciendo fork\n");
17            exit(EXIT_FAILURE);
18         }
19         else if(pid == 0){
20            printf ("PADRE %d \n",getppid());
21            printf("HIJO  %d \n",getpid());
22            break;
23         }
24      }
25   }
26
27   while(wait(NULL)>=0);
28   exit(EXIT_SUCCESS);
29 }
```

La clave de este ejercicio está en darse de cuenta que si la variable pid es igual a cero (en el momento que imprime su PID y el de su padre), es que está funcionando en ese momento como un proceso hijo, por lo que no puede hacer más forks, de ahí la sentencia `break;`.

Otro elemento de gran peso en el programa es en la línea 28 la instrucción `while(wait(NULL)>=0);`, que es la encargada de que el proceso padre espere por **todos** sus procesos hijos a que finalicen. Esto es así porque wait() devuelve un número negativo cuando no tiene hijos ejecutándose, por lo que el while realiza la función de esperar a que todos hayan termiando.

A continuación se muestra la salida por la terminal y el árbol generado.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio5b
PADRE 747
HIJO  748
PADRE 747
HIJO  749
PADRE 747
HIJO  750
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
```

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum$ pstree -p
init(1)─┬─bash(2)───pstree(751)
        └─bash(34)───ejercicio5b(747)─┬─ejercicio5b(748)
                                      ├─ejercicio5b(749)
                                      └─ejercicio5b(750)
```

En el árbol se puede ver perfectamente que un padre origina un conjunto de procesos hijos.

Además, en la salida de la terminal podemos ver que el PID del padre es siempre el mismo.

6 Ejercicio 6

Este ejercicio tiene la misión de enseñar como funciona la memoria dinámica con procesos de por medio. El ejercicio pedía que el proceso padre reservase memoria dinámica para una estructura que albergase una cadena de 80 caracteres y un entero. A continuación debía generar un proceso hijo y que este pidiese al usuario un nombre para guardar en la estructura, para luego comprobar si el proceso padre tenía acceso a esa cadena guardada en el hijo.

Mostramos el código adoptado a continuación.

Código ejercicio 6

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  #include <unistd.h>
7  #include <errno.h>
8
9  #define LEN 80
10
11 typedef struct{
12   char cad[LEN];
13   int n;
14 }Estruct;
15
16
17 int main(){
18   Estruct *est=NULL;
19   int pid;
20
21   est = (Estruct *) malloc(sizeof(Estruct));
22   if(!est){
23      printf("%s\n", strerror(errno));
24      return EXIT_FAILURE;
25   }
26
27   if(strcpy(est->cad, "TEXTO GUARDADO ANTES DE FORK\n")==NULL)
28      printf("%s\n", strerror(errno));
29      return EXIT_FAILURE;
30   }
```

```
31
32    pid = fork();
33
34    if(!pid){
35       printf("Introduzca un nombre de maximo %d caracteres: ",
36              LEN-1);
37       fgets(est->cad, LEN-1, stdin);
38    }
39
40    wait(NULL);
41
42    printf("%s", est->cad);
43
44    free(est);
45
46    return 0;
47 }
```

El código en el fichero fuente se encuentra comentado a diferencia del mostrado aquí.
El principio del programa es trivial, declaración de la estructura y reserva de memoria para la misma. A continuación se guarda en la cadena el texto "TEXTO GUARDADO ANTES DEL FORK" para poder ver que se imprime en el proceso padre después de que en el hijo se haya sobreescrito la cadena con el nombre itroducido.

Como era de esperar, se realiza el fork, si pid==0 estamos en el proceso hijo y por lo tanto pedimos al usuario un nombre, el cual guardamos en la cadena de la estructura. Una ves de vuelta en el proceso padre (especial atención al wait() para esperar al proceso hijo) se imprime el contenido de la cadena por pantalla.

Se muestra ahora la salida del programa.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
==853== Memcheck, a memory error detector
==853== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==853== Using Valgrind-3.14.0.SVN and LibVEX; rerun with -h for copyright info
==853== Command: ./ejercicio6
==853==
Introduzca un nombre de maximo 79 caracteres: NombreAleatorio
NombreAleatorio
==854==
==854== HEAP SUMMARY:
==854==     in use at exit: 0 bytes in 0 blocks
==854==   total heap usage: 3 allocs, 3 frees, 8,276 bytes allocated
==854==
==854== All heap blocks were freed -- no leaks are possible
==854==
==854== For counts of detected and suppressed errors, rerun with: -v
==854== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
TEXTO GUARDADO ANTES DEL FORK
==853==
==853== HEAP SUMMARY:
==853==     in use at exit: 0 bytes in 0 blocks
==853==   total heap usage: 2 allocs, 2 frees, 4,180 bytes allocated
==853==
==853== All heap blocks were freed -- no leaks are possible
==853==
==853== For counts of detected and suppressed errors, rerun with: -v
==853== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
```

Esto contesta a varias de las preguntas. Se puede inferir que las modificaciones adoptadas en el proceso hijo en la cadana no interfiere en la cadena del proceso padre, por lo que se puede deducir que al hacer un fork la memoria reservada es duplicada, una zona de memoria contendrá las variables del proceso padre y otra zona para los datos del hijo. Después del wait(), cada uno imprime su contenido, dando como resultado que la estructura del padre no ha recibido modificaciones y la del hijo tiene la cadena introducida por pantalla.

Por último, en el programa se ha liberado la memoria reservada en ambos procesos debido a que, como se ha dicho, cuando se hace fork se duplica la memoria reservada y cada proceso continua accediendo a la suya. Se adjunta en la imagen el reporte de memoria de Valgrind, el cual ratifica lo dicho anteriormente.

7 Ejercicio 8

Este es uno de los ejercicios más complejos y enriquecedores de la práctica en nuestra opinión. Junto los forks y waits con una nueva familia de funciones desconocida para nosotros; y este ejercicio nos ha ayudado en gran medida a comprender su utilidad y funcionamiento.

Se nos pide escribir un programa que cree tantos procesos hijos como programas ejecutables reciba el proceso padre como argumentos de entrada. Además, dependiendo del último argumento de entrada, se llamará a la ejecución de estos programas usando cuatro funciones diferentes de la familia exec() descritas en la documentación de la práctica.

A pesar de que se especifica en el enunciado que el orden de ejecución es desde el primero al último, hemos hecho otro segundo programa casi idéntico al primero que en lugar de

ejecutar desde el principio hasta el programa n, empieza ejecutando el programa n-ésimo hasta el primero pasado como argumento de entrada. Son respectivamente, los programas denominados como `ejercicio8_1.c` y `ejercicio8_2-c`.
Debajo se muestra el código simplicado (sin comentarios) del programa adoptado.

Código ejercicio 8_1

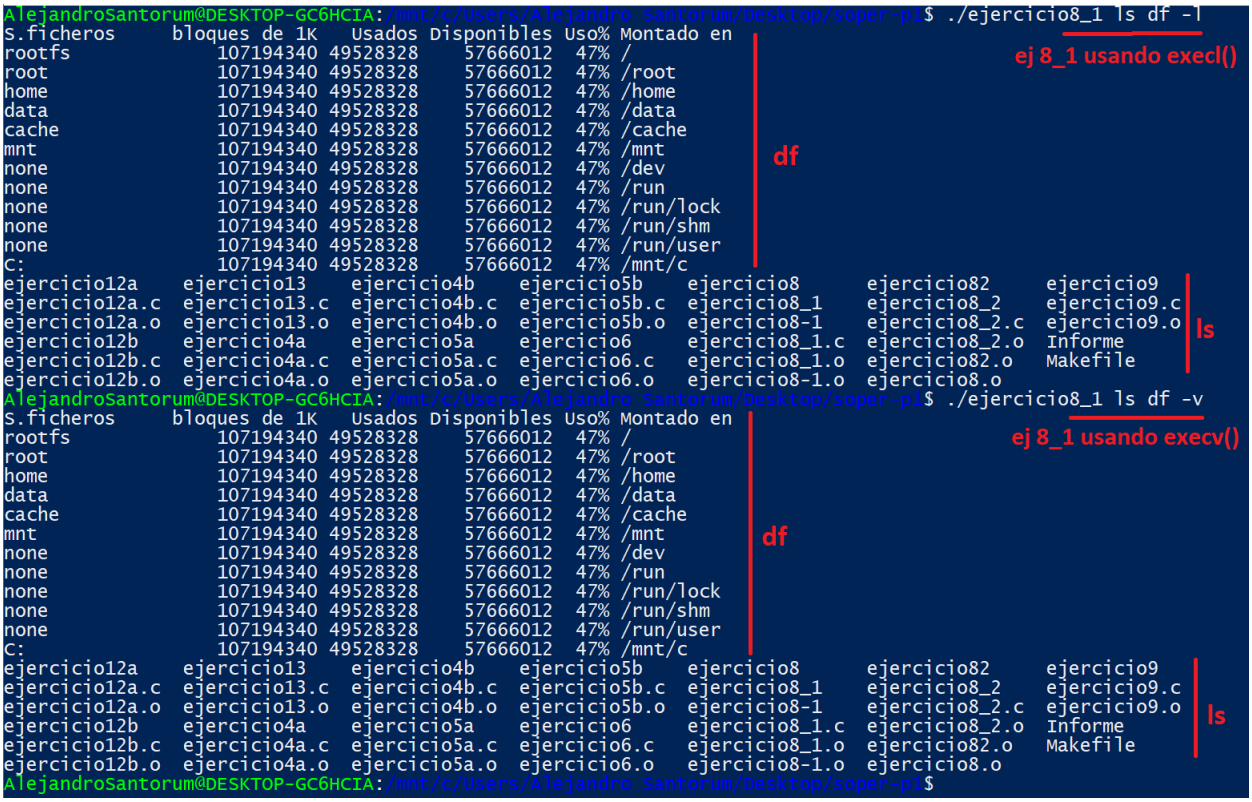
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  #include <unistd.h>
7  #include <errno.h>
8
9  #define PATH_LEN 100
10
11 int main(int argc, char **argv){
12  int pid, flag, i, j, k;
13  char **argum=NULL;
14  char path[PATH_LEN];
15
16  if(argc < 3){
17   printf("Error. Parametros de entrada insuficientes.\n");
18   printf("Introduzca nombres de los programas ejecutables y
19         despues\n");
20   printf("la indicacion de la funcion exec que desea
21          utilizar(-l, -lp, -v, -vp).\n");
22   return EXIT_FAILURE;
23  }
24
25  if(strcmp(argv[argc-1], "-l") && strcmp(argv[argc-1], "-lp")
26     && strcmp(argv[argc-1], "-v") &&
27     strcmp(argv[argc-1], "-vp")){
28   printf("Error. Indicacion de la funcion exec a utilizar
29          desconocida.\n");
30   printf("Indicacion introducida: %s\n", argv[argc-1]);
31   return EXIT_FAILURE;
32  }
33
34  if(argc > 3){
35   if((pid=fork()) <0 ){
36    printf("Error al emplear fork\n");
37    return EXIT_FAILURE;
38   }
39   flag=1;
```

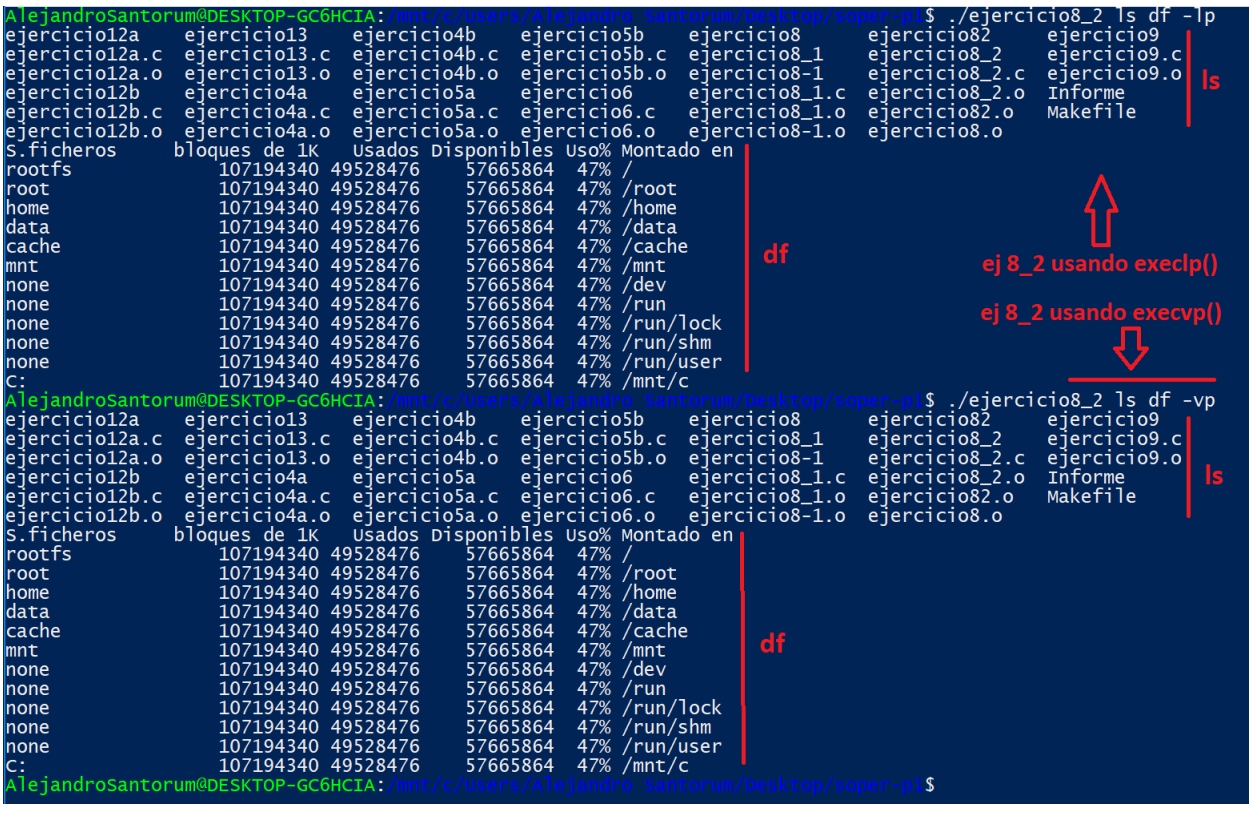
```
40
41   if(!pid){
42    argum = (char **) malloc(argc * sizeof(char *));
43    if(argum==NULL){
44     printf("%s\n", strerror(errno));
45     exit(EXIT_FAILURE);
46    }
47
48    if((argum[0] = strcpy(((char *)
49        malloc((strlen(argv[0])+1)*sizeof(char))),
50        argv[0])) == NULL){
51     free(argum);
52     printf("Error de memoria.\n");
53     exit(EXIT_FAILURE);
54    }
55    for(i=2, j=1; i<argc; i++, j++){
56     if((argum[j] = strcpy(((char *)
57         malloc((strlen(argv[i])+1)*sizeof(char))),
58         argv[i])) == NULL){
59      for(k=0;- k<j; k++){
60       free(argum[k]);
61      }
62      free(argum);
63      printf("Error de memoria.\n");
64      exit(EXIT_FAILURE);
65     }
66    }
67    argum[argc-1] = NULL;
68
69    execvp(argv[0], argum);
70    perror("Fallo en exec llamando de nuevo al programa
71           padre.\n");
72    exit(EXIT_FAILURE);
73   }
74  }
75
76  if(flag) wait(NULL);
77
78  if(!strcmp(argv[argc-1], "-l")){
79   sprintf(path, "/bin/bash");
80
81   execl(path, "bash", "-c", argv[1], NULL);
82  }
83
84  else if(!strcmp(argv[argc-1], "-lp")){
```

```
85   execlp(argv[1], argv[1], NULL);
86  }
87
88  else if(!strcmp(argv[argc-1], "-v")){
89   sprintf(path, "/bin/bash");
90
91   argum = (char **) malloc(4 * sizeof(char *));
92   if(argum==NULL){
93    printf("Error de memoria.\n");
94    exit(EXIT_FAILURE);
95   }
96   argum[0] = strcpy(((char *)
97                      malloc((strlen("bash")+1)*sizeof(char))),
98                      "bash");
99   if(argum[0]==NULL){ /* Error reservando memoria */
100   free(argum);
101   printf("Error de memoria.\n");
102   exit(EXIT_FAILURE);
103  }
104  argum[1] = strcpy(((char *)
105                     malloc((strlen("-c")+1)*sizeof(char))),
106                     "-c");
107  if(argum[1]==NULL){
108   free(argum[0]);
109   free(argum);
110   printf("Error de memoria.\n");
111   exit(EXIT_FAILURE);
112  }
113  argum[2] = strcpy(((char *)
114                     malloc(strlen(argv[1])*sizeof(char))),
115                     argv[1]);
116  if(argum[2]==NULL){
117   free(argum[0]);
118   free(argum[1]);
119   free(argum);
120   printf("Error de memoria.\n");
121   exit(EXIT_FAILURE);
122  }
123  argum[3] = NULL;
124
125  execv(path, argum);
126 }
127
128 else if(!strcmp(argv[argc-1], "-vp")){
129  argum = (char **) malloc(2 * sizeof(char *));
```

```
130   if(argum==NULL){
131    printf("Error de memoria.\n");
132    exit(EXIT_FAILURE);
133   }
134   argum[0] = strcpy(((char *)
135                      malloc(strlen(argv[1])*sizeof(char))),
136                      argv[1]);
137   if(argum[0]==NULL){
138    free(argum);
139    printf("Error de memoria.\n");
140    exit(EXIT_FAILURE);
141   }
142   argum[1] = NULL;
143
144   execvp(argv[1], argum);
145  }
146
147  else{
148   printf("Error inverosimil, esta comprobacion ya habia sido
149           superada anteriormente.\n");
150   return EXIT_FAILURE;
151  }
152
153  printf("\nNo se ha ejecutado el programa %s utilizando
154          %s\n\n", argv[1], argv[argc-1]);
155
156  return 0;
157 }
```

Se puede resumir la funcionalidad del código de la siguiente forma. Primero, se comprueban los parámetros de entrada del programa principal buscando que no haya nada inusual. Después, si hay más de dos programas aún por ejecutar (es decir, los argumentos de entrada ¿ 3) se crea un proceso hijo, el cual vuelve a llamar al programa principal usando una función de la familia exec(), pero esta vez con un número de programas $n-1$ siendo n el número de programas que se han especificado para ejecutar en ese momento.
Por último, el programa padre continua con el objetivo de ejecutar el programa destinado con una funcion del tipo exec() especificada por línea de comandos.
Justo debajo enseñamos las salidas de dicho/s programa/s.


AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio8_1 ls df -l
ej 8_1 usando execl()
df
ls
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio8_1 ls df -v
ej 8_1 usando execv()
df
ls
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$


AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio8_2 ls df -lp
ls
df
ej 8_2 usando execlp()
ej 8_2 usando execvp()
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio8_2 ls df -vp
ls
df
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$

8 Ejercicio 9

Si hemos dicho que el ejercicio anterior era completo y enriquecedor, este no se queda atrás. Consiste en aprender como comunicar dos o más procesos entre si con la ayuda de las tuberías o pipes.

El problema pedía que se crearan cuatro procesos hijos que, con los operandos adquiridos en el proceso padre y pasados a sus hijos por tuberías, realizasen cada uno de ellos diferentes operaciones y devolviesen al proceso padre el resultado mediante otras tuberías. A continuación se muestra el código del ejercicio sin comentarios.

Código ejercicio 9

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8  #include <errno.h>
9
10 #define LEER 0
11 #define ESCRIBIR 1
12 #define N_HIJOS 4
13 #define LEN 200
14
15 int separar_primero(char *str);
16
17 int separar_segundo(char *str);
18
19 int factorial(int a);
20
21 int abs(int a);
22
23 int main(){
24  int fd[2*N_HIJOS][2];
25  int fpid, h, i;
26  int ids[4];
27  int O1, O2;
28  float O3;
29  char str_write[LEN], str_read[LEN];
30
31  fpid = getpid();
32
33  for(h=0; h<2*N_HIJOS; h++){
```

```
34   if(pipe(fd[h]) == -1){
35    printf("Error creando pipe numero %d", h);
36    exit(EXIT_FAILURE);
37   }
38  }
39
40  for(h=0; h<N_HIJOS; h++){
41   if(getpid() == fpid){
42    if((ids[h] = fork()) < 0){
43     printf("Error. Fork numero %d", h);
44     exit(EXIT_FAILURE);
45    }
46    if(getpid() != fpid){
47     break;
48    }
49   }
50  }
51
52  if(getpid() == fpid){
53   fflush(stdin);
54   printf("Introduzca el primer operando: ");
55   scanf("%d", &O1);
56   fflush(stdin);
57   printf("Introduzca el segundo operando: ");
58   scanf("%d", &O2);
59
60   sprintf(str_write, "%d,%d", O1, O2);
61
62   for(i=0; i<N_HIJOS; i++){
63    close(fd[i][LEER]);
64    write(fd[i][ESCRIBIR], str_write, strlen(str_write));
65   }
66  }
67
68  if(h == 0){
69   memset(str_read,0,LEN);
70   memset(str_write,0,LEN);
71   close(fd[h][ESCRIBIR]);
72   read(fd[h][LEER], str_read, LEN);
73
74   O1 = separar_primero(str_read);
75   O2 = separar_segundo(str_read);
76
77   O3 = pow(O1,O2);
78
```

```
 sprintf(str_write, "Datos enviados a traves de la tuberia
        por el proceso PID=%d. Operando 1: %d. Operando 2:
        %d. Potencia: %.2f \n", getpid(), O1, O2, O3);

 close(fd[h+4][LEER]);
 write(fd[h+4][ESCRIBIR], str_write, strlen(str_write));
}

if(h == 1){
 memset(str_read,0,LEN);
 memset(str_write,0,LEN);
 close(fd[h][ESCRIBIR]);
 read(fd[h][LEER], str_read, LEN);

 O1 = separar_primero(str_read);
 O2 = separar_segundo(str_read);
 if(O1<0 || O2<0){
  sprintf(str_write,"Datos enviados a traves de la tuberia
        por el proceso PID=%d. Operando 1: %d. Operando 2:
        %d. No se puede hacer el factorial \n", getpid(),
        O1, O2);
 }

 else{
  O3 = (float)factorial(O1)/(float)(O2);

  sprintf(str_write,"Datos enviados a traves de la tuberia
        por el proceso PID=%d. Operando 1: %d. Operando 2:
        %d. Factorial entre el numero: %.2f \n", getpid(),
        O1, O2, O3);
 }

 close(fd[h+4][LEER]);
 write(fd[h+4][ESCRIBIR], str_write, strlen(str_write));
}

if(h == 2){
 memset(str_read,0,LEN);
 memset(str_write,0,LEN);
 close(fd[h][ESCRIBIR]);
 read(fd[h][LEER], str_read, LEN);

 O1 = separar_primero(str_read);
 O2 = separar_segundo(str_read);

```

```
124   if(O1<O2 || O1<0 || O2<0){
125    sprintf(str_write,"Datos enviados a traves de la tuberia
126            por el proceso PID=%d. Operando 1: %d. Operando 2:
127            %d. No hay permutacion posible \n", getpid(),
128            O1, O2);
129   }
130   else{
131    O3 = (float)factorial(O1)/(float)factorial(O2);
132
133    sprintf(str_write,"Datos enviados a traves de la tuberia
134            por el proceso PID=%d. Operando 1: %d. Operando 2:
135            %d. Permutacion: %.2f \n", getpid(), O1, O2, O3);
136   }
137
138   close(fd[h+4][LEER]);
139   write(fd[h+4][ESCRIBIR], str_write, strlen(str_write));
140  }
141
142  if(h == 3){
143   memset(str_read,0,LEN);
144   memset(str_write,0,LEN);
145   close(fd[h][ESCRIBIR]);
146   read(fd[h][LEER], str_read, LEN);
147
148   O1 = separar_primero(str_read);
149   O2 = separar_segundo(str_read);
150
151   O3 = abs(O1)+abs(O2);
152
153   sprintf(str_write,"Datos enviados a traves de la tuberia por
154           el proceso PID=%d. Operando 1: %d. Operando 2: %d.
155           Suma de valores absolutos: %.2f \n", getpid(), O1,
156           O2, O3);
157
158   close(fd[h+4][LEER]);
159   write(fd[h+4][ESCRIBIR], str_write, strlen(str_write));
160  }
161
162  if(fpid == getpid()){
163   for(h=0; h < N_HIJOS; h++){
164    memset(str_read,0,LEN);
165    close(fd[h+4][ESCRIBIR]);
166    read(fd[h+4][LEER], str_read, LEN);
167    printf("%s",str_read);
168   }
```

```
169  }
170  exit(EXIT_SUCCESS);
171 }
172
173
174 int separar_primero(char *str){
175  char *copia, *aux;
176  int final;
177
178  aux = (char *)malloc(sizeof(char)*LEN);
179  if(aux == NULL){
180   printf("Error reservando memoria\n");
181   exit(EXIT_FAILURE);
182  }
183  copia = strcpy((char *)
184                 malloc(sizeof(char)*sizeof(str+1)),str);
185  aux = strchr(copia, ',');
186  *aux = 0;
187  final = atoi(copia);
188  free(copia);
189  return final;
190 }
191
192
193 int separar_segundo(char *str){
194  char *aux;
195  int final;
196
197  aux = (char *)malloc(sizeof(char)*LEN);
198  if(aux == NULL){
199   printf("Error reservando memoria\n");
200   exit(EXIT_FAILURE);
201  }
202  aux = strchr(str, ',');
203  aux++;
204  final = atoi(aux);
205  return final;
206 }
207
208
209 int factorial(int a){
210  if(a<0){
211   printf("Error al calcular el factorial, el numero es
212          negativo.\n");
213   return -1;
```

```
214   }
215
216   if(a==0 || a==1){
217    return 1;
218   }
219
220   return a*factorial(a-1);
221  }
222
223
224  int abs(int a){
225   if(a<0){
226    return -a;
227   }
228   return a;
229  }
```

Vamos a explicar el programa. Lo primero que llama la atención es la declaración fd[2*N_HIJOS][2]. Lo que se está haciendo es preparar 16 enteros que serán nuestras futuras tuberías, que se crearán con la función `pipe(fd[h])`. ¿Por qué se necesitan 16? Pues bien, cada hijo tiene que comunicarse con su padre. En primer lugar, es el padre que envía (escribe) a cada hijo (lee) los operandos a utilizar, por lo que tendrán que cerrar los extremos de las tuberías correspondientes, el padre cierra lectura y el hijo escritura.
Una vez se hayan hecho las operaciones, los procesos cambian los papeles, los hijos envían el resultado (escriben) al proceso padre (lee), por lo que los hijos tendrán que cerrar el extremo de lectura y el padre el de escritura. A lo largo de este proceso se han necesita 4 tuberías por cada par padre-hijo, como tenemos 4 hijos (4 operaciones a realizar) resulta en un total de 16 tuberías.

Después de lo descrito anteriormente, no queda mucho que explicar. Simplemente contar que se utilizan las funciones de bajo nivel para control de ficheros `read()` y `write()`, pues las tuberías son en realidad ficheros de uso compartido por el proceso padre y su proceso hijo.

De seguido, mostramos tres ejecuciones del programa con operandos diferentes.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio9
Introduzca el primer operando: -2
Introduzca el segundo operando: 5
Datos enviados a través de la tubería por el proceso PID=184. Operando 1: -2. Operando 2: 5. Potencia: -32.00
Datos enviados a través de la tubería por el proceso PID=185. Operando 1: -2. Operando 2: 5. No se puede hacer el factorial
Datos enviados a través de la tubería por el proceso PID=186. Operando 1: -2. Operando 2: 5. No hay permutacion posible
Datos enviados a través de la tubería por el proceso PID=187. Operando 1: -2. Operando 2: 5. Suma de valores absolutos: 7.00
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio9
Introduzca el primer operando: 3
Introduzca el segundo operando: 5
Datos enviados a través de la tubería por el proceso PID=189. Operando 1: 3. Operando 2: 5. Potencia: 243.00
Datos enviados a través de la tubería por el proceso PID=190. Operando 1: 3. Operando 2: 5. Factorial entre el numero: 1.20
Datos enviados a través de la tubería por el proceso PID=191. Operando 1: 3. Operando 2: 5. No hay permutacion posible
Datos enviados a través de la tubería por el proceso PID=192. Operando 1: 3. Operando 2: 5. Suma de valores absolutos: 8.00
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio9
Introduzca el primer operando: 5
Introduzca el segundo operando: 3
Datos enviados a través de la tubería por el proceso PID=194. Operando 1: 5. Operando 2: 3. Potencia: 125.00
Datos enviados a través de la tubería por el proceso PID=195. Operando 1: 5. Operando 2: 3. Factorial entre el numero: 40.00
Datos enviados a través de la tubería por el proceso PID=196. Operando 1: 5. Operando 2: 3. Permutacion: 20.00
Datos enviados a través de la tubería por el proceso PID=197. Operando 1: 5. Operando 2: 3. Suma de valores absolutos: 8.00
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
```

Podemos observar que las operaciones son las correctas, siempre y cuando se puedan realizar. Se ejemplifica esto último en las ejecuciones número 1 y número 2.
En la primera no se ha podido calcular el factorial porque uno de los números es negativo (pasaría lo mismo si 02 fuera menor que cero), ni tampoco las permutaciones por el mismo motivo.
En la segunda sí que se ha podido calcular el factorial, pero no las permutaciones porque el operando número 1 era mayor que el número 2.
En la ejecución número 3 no hay errores con los operandos, por lo que las operaciones son realizadas correctamente.

9 Ejercicio 12-A

Este problema se enfoca en calcular el tiempo de ejecución de un determinado programa. En el ejercicio 12a se realizará el cálculo de los 10000 primeros primos en 100 procesos hijos y se medirá lo que ha tardado para posteriormente, en el ejercicio que precede a este, el ejercicio 12b, compararlo con el tiempo que han necesitado 100 hilos diferentes en calcular también los 10000 primeros primos.
Sin más preámbulos, se muestra el código a continuación.

Código ejercicio 12a

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

#define LEN 100
```

```
14 #define TENTOTHENINE 1000000000
15 #define N_HIJOS 100
16
17 typedef struct{
18  char cad[LEN];
19  int n;
20 } Estruct;
21
22
23 int esPrimo(int n);
24
25
26 int* calcularPrimos(int n_primos);
27
28
29 int main(int argc, char const *argv[]) {
30  int pid, i, check1, check2;
31  int *primos=NULL;
32  Estruct *est=NULL;
33  double total_time=0;
34  struct timespec start, end, aux;
35
36  if(argc != 2){
37   printf("Parametros de entrada erroneos.\n");
38   printf("Introduzca nombre del programa y el numero de primos
39          a calcular.\n");
40   exit(EXIT_FAILURE);
41  }
42
43  est = (Estruct *) malloc(sizeof(Estruct));
44  if(!est){
45   printf("%s\n", strerror(errno));
46   return EXIT_FAILURE;
47  }
48
49  check1 = clock_gettime(CLOCK_REALTIME, &start);
50
51  for(i=0; i<N_HIJOS; i++){
52   if((pid = fork()) < 0){
53    printf("Error haciendo fork en la iteracion numero %d", i);
54    free(est);
55    exit(EXIT_FAILURE);
56   }
57   if(!pid) break;
58  }
```

```
59  if(!pid){
60   primos = calcularPrimos(atoi(argv[1]));
61   if(!primos){
62    printf("Error reservando memoria para el array.\n");
63    free(est);
64    exit(EXIT_FAILURE);
65   }
66   free(primos);
67
68   exit(EXIT_SUCCESS);
69  }
70  while(wait(NULL) >= 0);
71
72  check2 = clock_gettime(CLOCK_REALTIME, &end);
73
74  if(check1==-1){
75   printf("Error calculando el tiempo inicial.\n");
76   free(est);
77   exit(EXIT_FAILURE);
78  }
79  if(check2==-1){
80   printf("Error calculando el tiempo final.\n");
81   free(est);
82   exit(EXIT_FAILURE);
83  }
84
85  aux.tv_sec = end.tv_sec - start.tv_sec;
86  aux.tv_nsec = end.tv_nsec - start.tv_nsec;
87  total_time = total_time + aux.tv_sec*TENTOTHENINE +
88              aux.tv_nsec;
89
90  printf("El tiempo necesitado para crear %d procesos y calcular
91         los\n", N_HIJOS);
92  printf("%d primos primos en cada uno de ellos ha sido de:\n",
93         atoi(argv[1]));
94  printf("%0.1f nanosegundos = %0.4f segundos\n", total_time,
95         (total_time/TENTOTHENINE));
96
97  free(est);
98
99  exit(EXIT_SUCCESS);
100 }
101
102
103
```

```
104 int esPrimo(int n){
105  int k;
106
107  for(k=2; k<=sqrt(n); k++){
108   if(!(n % k)) return 0;
109  }
110  return 1;
111 }
112
113
114
115 int* calcularPrimos(int n_primos){
116  int *primos=NULL;
117  int contador=0, i;
118
119  primos = (int *) malloc(n_primos * sizeof(int));
120  if(!primos){
121   printf("Error de memoria.\n");
122   return NULL;
123  }
124
125  primos[0] = 2;
126  contador++;
127
128  i = 3;
129  while(contador < n_primos){
130   if(esPrimo(i)){
131    primos[contador] = i;
132    contador++;
133   }
134  }
135  return primos;
136 }
```

No hay mucho que comentar. Se han generado 100 procesos hijos a partir de un único proceso padre (al estilo del ejercicio 5b), y cada uno de ellos ha calculado los n primeros primos, siendo n pasado como argumento de entrada al programa principal. Para nuestros ejemplos, será 10000.
La salida de la ejecución de este programa será enseñada y comentada en el ejercicio 12b.

10 Ejercicio 12-B

Como se ha comentado anteriormente, en este ejercicio vamos a generar 100 hilos y en cada uno de ellos calcular los 10000 primeros primos, midiendo su tiempo y comparándolo con el

mismo ejercicio pero con 100 procesos.
El código utilizando threads o hilos se muestra debajo.

Código ejercicio 12b

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  #include <unistd.h>
7  #include <errno.h>
8  #include <pthread.h>
9  #include <limits.h>
10 #include <time.h>
11 #include <math.h>
12
13 #define LEN 100
14 #define TENTOTHENINE 1000000000
15 #define N_HILOS 100
16
17 typedef struct{
18  char cad[LEN];
19  int n;
20 } Estruct;
21
22
23 int esPrimo(int n);
24
25
26 void* calcularPrimos(void *arg);
27
28
29 int main(int argc, char const *argv[]) {
30  pthread_t hilo[N_HILOS];
31  Estruct *est=NULL;
32  double total_time=0;
33  struct timespec start, end, aux;
34  int i, check1, check2;
35
36  if(argc != 2){
37   printf("Parametros de entrada erroneos.\n");
38   printf("Introduzca nombre del programa y el numero de primos
39          a calcular.\n");
40   exit(EXIT_FAILURE);
41  }
```

```
42
43  est = (Estruct *) malloc(sizeof(Estruct));
44  if(!est){
45   printf("%s\n", strerror(errno));
46   return EXIT_FAILURE;
47  }
48
49  est->n = atoi(argv[1]);
50
51  check1 = clock_gettime(CLOCK_REALTIME, &start);
52
53  for(i=0; i<N_HILOS; i++){
54   pthread_create(&hilo[i], NULL, calcularPrimos, (void *)est);
55  }
56
57  for(i=0; i<N_HILOS; i++){
58   pthread_join(hilo[i], NULL);
59  }
60
61  check2 = clock_gettime(CLOCK_REALTIME, &end);
62
63  if(check1==-1){
64   printf("Error calculando el tiempo inicial.\n");
65   free(est);
66   exit(EXIT_FAILURE);
67  }
68  if(check2==-1){
69   printf("Error calculando el tiempo final.\n");
70   free(est);
71   exit(EXIT_FAILURE);
72  }
73
74  aux.tv_sec = end.tv_sec - start.tv_sec;
75  aux.tv_nsec = end.tv_nsec - start.tv_nsec;
76  total_time = total_time + aux.tv_sec*TENTOTHENINE +
77               aux.tv_nsec;
78
79  printf("El tiempo necesitado para crear %d HILOS y calcular
80         los\n", N_HILOS);
81  printf("%d primos primos en cada uno de ellos ha sido de:\n",
82         atoi(argv[1]));
83  printf("%0.1f nanosegundos = %0.4f segundos\n", total_time,
84         (total_time/TENTOTHENINE));
85
86  free(est);
```

```
87
88  exit(EXIT_SUCCESS);
89
90  return 0;
91 }
92
93
94
95 int esPrimo(int n){
96  int k;
97
98  for(k=2; k<=sqrt(n); k++){
99   if(!(n % k)) return 0;
100  }
101  return 1;
102 }
103
104
105
106 void* calcularPrimos(void *arg){
107  Estruct *est;
108  int *primos=NULL;
109  int contador=0, i;
110  int n;
111
112  if(!arg){
113   printf("Error de memoria. Estructura de argumentos de
114           entrada a NULL.\n");
115   return NULL;
116  }
117
118  est = (Estruct *) arg;
119  n = est->n;
120
121  primos = (int *) malloc(n * sizeof(int));
122  if(!primos){
123   printf("Error de memoria.\n");
124   return NULL;
125  }
126
127  primos[0] = 2;
128  contador++;
129
130  i = 3;
131  while(contador < n){
```

```
132     if(esPrimo(i)){
133       primos[contador] = i;
134       contador++;
135     }
136   }
137   free(primos);
138   return NULL;
139 }
```

Comentemos ahora un poco el programa. Para empezar, la función que crea los hilos es `pthread_create(...)`, que es lo análogo con los procesos a `fork()`. Del mismo modo, `pthread_join(...)` cumple la misión de `wait()`.
El constructor de hilos tiene como argumetos la dirección de un hilo o pthread_t, de segundo los atributos de la creación del hilos, que si es NULL se escogerán por defecto. De tercero el nombre de la función que será llamada automáticamente en el hilo y, por último, el argumento void* que recibe la función que va a ser llamada. Una técnica muy usada es pasar todos los argumentos que se quieran mediante una estructura "casteada" a void*.

Mostremos la salida que origina este programa y el anterior para comparar los tiempos de ejecución.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio12a 10000
El tiempo necesitado para crear 100 procesos y calcular los
10000 primos primos en cada uno de ellos ha sido de:
90103300.0 nanosegundos = 0.0901 segundos
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio12b 10000
El tiempo necesitado para crear 100 HILOS y calcular los
10000 primos primos en cada uno de ellos ha sido de:
11717500.0 nanosegundos = 0.0117 segundos
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
```

En este ejemplo en particular, el tiempo utilizando procesos ha sido de 0.09 segundos y el mismo programa utilizando hilos ha sido de 0.011 segundos.
El tiempo usando hilos ha sido prácticamente 9 veces inferior, como cabía de esperar, pues es mucho más eficiente la creación, comunicación y eliminación de hilos que de procesos.

Esto es debido a que la creación de los hilos es mucho más eficiente que la de los procesos, ya que necesitan copiar muy poca memoria, solo la pila del propio hilo, a diferencia de los procesos, que tenían que duplicar toda la memoria reservada/utilizada.
Por otro lado, es mucho más rápido cambiar los estados de Listo-Ejecución entre hilos que entre procesos.
Por último, la comunicación de hilos es mucho más rápida que la de los procesos, ya que comparten la memoria común del proceso.

11 Ejercicio 13

Llegamos al último ejercicio de la práctica 1. Esta vez se nos pide calcular el producto de dos escalares por dos matrices en dos hilos trabajando en paralelo.

La secuencia del programa es pedir la dimesión de las matrices al usuario, para después preguntarle por los dos escalares y las dos matrices. Después de esto, dos hilos paralelos calcularán el producto de un escalar por una matriz cuadrada cada uno, mostrando las operaciones cada vez que acaban con una fila.
A continuación se muestra el código del programa.

Código ejercicio 13

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  #include <unistd.h>
7  #include <errno.h>
8  #include <pthread.h>
9  #include <limits.h>
10 #include <time.h>
11 #include <math.h>
12
13 #define LEN 50
14
15 typedef struct{
16  int id;
17  int **matrix;
18  int dim;
19  int scalar;
20  char *pos1;
21  char *pos2;
22 } Argum;
23
24
25
26 void* mult_matrix(void *arg);
27
28
29 int main(int argc, char const *argv[]) {
30  int **m1=NULL;
31  int **m2=NULL;
32  int s1;
33  int s2;
34  int i, j, dim;
35  char comunic1[LEN];
36  char comunic2[LEN];
37  Argum *arg1=NULL;
38  Argum *arg2=NULL;
```

```
39  pthread_t h1, h2;
40
41  printf("Introduzca dimension de la matriz cuadrada:\n");
42  scanf("%d", &dim);
43
44  printf("Introduzca multiplicador 1:\n");
45  scanf("%d", &s1);
46  printf("Introduzca multiplicador 2:\n");
47  scanf("%d", &s2);
48
49  m1 = (int **) malloc(dim * sizeof(int *));
50  if(!m1){
51   printf("Error de memoria matriz 1.\n");
52   exit(EXIT_FAILURE);
53  }
54  for(i=0; i<dim; i++){
55   m1[i] = (int *) malloc(dim * sizeof(int));
56   if(!m1[i]){
57    printf("Error de memoria m1-%d", i);
58    for(j=0; j<i; j++){
59     free(m1[j]);
60    }
61    free(m1);
62    exit(EXIT_FAILURE);
63   }
64  }
65
66  printf("MATRIZ 1\n");
67  for(i=0; i<dim; i++){
68   for(j=0; j<dim; j++){
69    printf("Introduzca elemento %d-%d: ", i+1,j+1);
70    scanf("%d", &(m1[i][j]));
71   }
72  }
73
74  m2 = (int **) malloc(dim * sizeof(int *));
75  if(!m2){
76   printf("Error de memoria matriz 2.\n");
77   exit(EXIT_FAILURE);
78  }
79  for(i=0; i<dim; i++){
80   m2[i] = (int *) malloc(dim * sizeof(int));
81   if(!m2[i]){
82    printf("Error de memoria m2-%d", i);
83    for(j=0; j<i; j++){
```

```
84      free(m2[j]);
85     }
86     free(m2);
87     exit(EXIT_FAILURE);
88    }
89   }
90
91   printf("MATRIZ 2\n");
92   for(i=0; i<dim; i++){
93    for(j=0; j<dim; j++){
94     printf("Introduzca elemento %d-%d: ", i+1,j+1);
95     scanf("%d", &(m2[i][j]));
96    }
97   }
98
99   arg1 = (Argum *) malloc(sizeof(Argum));
100  if(!arg1){
101   printf("Error de memoria estructura de comunicacion.\n");
102   exit(EXIT_FAILURE);
103  }
104  arg2 = (Argum *) malloc(sizeof(Argum));
105  if(!arg2){
106   printf("Error de memoria estructura de comunicacion.\n");
107   exit(EXIT_FAILURE);
108  }
109
110  strcpy(comunic1, " - el hilo 1 todavia no ha comenzado.\n");
111  strcpy(comunic2, " - el hilo 2 todavia no ha comenzado.\n");
112
113  printf("Realizando producto:\n");
114
115  arg1->id=1;
116  arg1->matrix = m1;
117  arg1->dim = dim;
118  arg1->scalar = s1;
119  arg1->pos1 = comunic1;
120  arg1->pos2 = comunic2;
121
122  pthread_create(&h1, NULL, mult_matrix, (void *)arg1);
123
124  arg2->id=2;
125  arg2->matrix = m2;
126  arg2->dim = dim;
127  arg2->scalar = s2;
128  arg2->pos1 = comunic1;
```

```
129  arg2->pos2 = comunic2;
130
131  pthread_create(&h2, NULL, mult_matrix, (void *)arg2);
132
133  pthread_join(h1, NULL);
134  pthread_join(h2, NULL);
135
136  for(i=0; i<dim; i++){
137   free(m1[i]);
138   free(m2[i]);
139  }
140  free(m1);
141  free(m2);
142  free(arg1);
143  free(arg2);
144
145  return EXIT_SUCCESS;
146 }
147
148
149
150 void* mult_matrix(void *arg){
151  Argum *com;
152  int fila, col, res;
153  char str[LEN], num[LEN/5], fin[LEN];
154
155  if(!arg){
156   printf("Error de memoria. Estructura de argumentos de
157          entrada a NULL.\n");
158   return NULL;
159  }
160
161  com = (Argum *) arg;
162
163  for(fila=0; fila<com->dim; fila++){
164   if((com->id % 2)){
165    sprintf(com->pos1, " - el hilo %d va por la fila %d\n",
166            com->id, fila);
167    strcpy(fin, com->pos2);
168   }
169   if(!(com->id % 2)){
170    sprintf(com->pos2, " - el hilo %d va por la fila %d\n",
171            com->id, fila);
172    strcpy(fin, com->pos1);
173   }
```

```
174
175     sprintf(str, "Hilo %d multiplicando fila %d resultado",
176             com->id, fila);
177     for(col=0; col<com->dim; col++){
178      res = com->scalar * com->matrix[fila][col];
179      sprintf(num, " %d", res);
180      strcat(str, num);
181     }
182     strcat(str, fin);
183     printf("%s", str);
184     fflush(stdout);
185     sleep(1);
186   }
187   return NULL;
188 }
```

Se puede resumir la estrategia seguida muy fácilmente. Se ha programado una función que será ejecutada por dos hilos al mismo tiempo, cuya única misión es imprimir las matrices pasadas por el argumento void* que apunta a una estructura con todos los datos que necesita. En el programa principal, main, simplemente se piden los datos al usuario y se guardan en los campos de la estructura. Pthread_create(...) hace el resto.
Comentar que se ha hecho el apartado final, en el que cada hilo puede conocer por donde va el otro. Mostramos ahora un ejemplo de su ejecución.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$ ./ejercicio13
Introduzca dimension de la matriz cuadrada:
3
Introduzca multiplicador 1:
2
Introduzca multiplicador 2:
10
MATRIZ 1
Introduzca elemento 1-1: 1
Introduzca elemento 1-2: 2
Introduzca elemento 1-3: 3
Introduzca elemento 2-1: 4
Introduzca elemento 2-2: 5
Introduzca elemento 2-3: 6
Introduzca elemento 3-1: 7
Introduzca elemento 3-2: 8
Introduzca elemento 3-3: 9
MATRIZ 2
Introduzca elemento 1-1: 10
Introduzca elemento 1-2: 11
Introduzca elemento 1-3: 12
Introduzca elemento 2-1: 13
Introduzca elemento 2-2: 14
Introduzca elemento 2-3: 15
Introduzca elemento 3-1: 16
Introduzca elemento 3-2: 17
Introduzca elemento 3-3: 18
Realizando producto:
Hilo 1 multiplicando fila 0 resultado 2 4 6 - el hilo 2 todavia no ha comenzado.
Hilo 2 multiplicando fila 0 resultado 100 110 120 - el hilo 1 va por la fila 0
Hilo 1 multiplicando fila 1 resultado 8 10 12 - el hilo 2 va por la fila 0
Hilo 2 multiplicando fila 1 resultado 130 140 150 - el hilo 1 va por la fila 1
Hilo 1 multiplicando fila 2 resultado 14 16 18 - el hilo 2 va por la fila 1
Hilo 2 multiplicando fila 2 resultado 160 170 180 - el hilo 1 va por la fila 2
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro Santorum/Desktop/soper-p1$
```

Antes que nada comentar una decisión de diseño: en la documentación de la práctica se ejemplificaba la introducción de las matrices tal que así:

```
Introduzca matriz 1:
3 5 4 6 2 2 1 2 3
Introduzca matriz 2:
3 5 5 3 2 2 2 2 3
```

A diferencia de nuestra forma, que pedimos elemento a elemento. Esto puede ser un poco más incómodo para quien utiliza la línea de comandos, pero mejora la eficiencia del programa al evitar tener que "tokenizar" la string que sería obtenida por el método del ejemplo de la documentación., pasando de un coste $O(n)$ siendo $n = dim^2$ a un coste $O(1)$ ya que el dato es introducido directamente en la matriz.

Dejando ya a parte el estilo de código, comentar la estrategia seguida para comunicar un hilo con otro. Hemos utilizado el hecho de que los hilos no duplican la memoria reservada como sí hacen los procesos para poder comunicarlos mediante dos punteros de tipo char*. Uno de ellos es destinado a la escritura del hilo 1 y lectura del hilo 2, y el otro a la lectura del hilo 1 y escritura del 2.
Mediante este método, podemos pasar información de por qué parte del cálculo del producto se encuentran e imprimirlo por pantalla, tal y como se muestra en la imagen.

12 Conclusión y comentarios finales

Ha sido una práctica larga y productiva, en la que hemos aprendido muchas cosas sobre la programación orientada a la multi-tarea mediante los procesos e hilos.

Los primeros ejercicios han tenido el objetivo de enseñar cómo funciona el fork() y las interaciones proceso padre-proceso hijo.
Después se ha enfocado más a la intercomunicación de procesos para poder intercambiarse información y/o datos.
Por último, hemos obtenido la intuición de qué es un hilo y sus diferencias (para bien o para mal) con los procesos.