

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

04-03-2018

Contents

1	Introducción	2
2	Ejercicio 4-A	2
3	Ejercicio 4-B	3
4	Ejercicio 5-A	4
5	Ejercicio 5-B	5
6	Ejercicio 6	6
7	Ejercicio 8	7
8	Ejercicio 9	9
9	Ejercicio 12	10
10	Ejercicio 13	11
11	Conclusión y comentarios finales	13

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. Anteriormente se incluía el código, pero en esta versión final se ha prescindido del mismo ya que se puede acceder a el en la carpeta de la entrega.

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 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 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 huérfano 506
Proceso huérfano 512
PADRE 506
Proceso huérfano 507
HIJO 510
PADRE 507
PADRE 508
Proceso huérfano 509
Proceso huérfano 516
Proceso huérfano 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 huérfano 510
PADRE 510
Proceso huérfano 511
Proceso huérfano 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.

Igual que en el ejercicio 4a, se pedía que cada hijo imprimiese 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 huérfano 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 `ps tree -p` para percibir el árbol de procesos que se origina. Aquí un ejemplo.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum$ ps tree -p
init(1)─bash(2)─ps tree(204)
        │
        └─bash(34)─ejercicio4a(182)─ejercicio4a(183)─ejercicio4a(186)─ejercicio4a(193)─ejercicio4a(197)
                                   │
                                   │   └─ejercicio4a(189)─ejercicio4a(194)
                                   │   └─ejercicio4a(191)─ejercicio4a(196)
                                   │
                                   └─ejercicio4a(184)─ejercicio4a(188)─ejercicio4a(195)
                                       │
                                       │   └─ejercicio4a(190)
                                       │   └─ejercicio4a(192)
                                       └─ejercicio4a(185)─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 predecir 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 varían los programas: en el primero, si pudiésemos ejecutar `ps tree` 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.

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 `ps tree -p` enseñada en el ejercicio anterior.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum$ ps tree -p
init(1)─bash(2)─ps tree(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-pl$ ./ejercicio5a
PADRE 632
HIJO 633
PADRE 633
HIJO 634
PADRE 634
HIJO 635
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/soper-pl$
```

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.

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 terminado.

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

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

```
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.

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 sobrescrito la cadena con el nombre introducido.

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 vez 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-pl$
==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-pl$
```

Esto contesta a varias de las preguntas. Se puede inferir que las modificaciones adoptadas en el proceso hijo en la cadena 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`.

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 son más de 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 función del tipo `exec()` especificada por línea de comandos.

Justo debajo enseñamos las salidas de dicho/s programa/s.

Antes de nada, es importante subrayar que los funciones `exec()` que requieran del path del programa a ejecutar se han implementado de una forma un tanto peculiar: se ejecuta el programa `bash` con un parámetro de entrada, el programa a correr. ¿El por qué? Pues esto se ha implementado así porque dependiendo del programa (`ls df du top clear...`) el path será diferente. Entonces teníamos dos opciones, pedirle al usuario que introdujera el path completo, o ejecutar `bash` con parámetro de entrada el programa a lanzar. Se ha optado por la segunda opción, haciendo que el usuario no se tenga que preocupar de nada.

```

AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/soper-p1$ ./ejercicio8_1 ls df -l
S.ficheros    bloques de 1K  Usados  Disponibles  Uso%  Montado en
rootfs        107194340  49528328  57666012    47%  /
root          107194340  49528328  57666012    47%  /root
home          107194340  49528328  57666012    47%  /home
data          107194340  49528328  57666012    47%  /data
cache         107194340  49528328  57666012    47%  /cache
mnt           107194340  49528328  57666012    47%  /mnt
none          107194340  49528328  57666012    47%  /dev
none          107194340  49528328  57666012    47%  /run
none          107194340  49528328  57666012    47%  /run/lock
none          107194340  49528328  57666012    47%  /run/shm
none          107194340  49528328  57666012    47%  /run/user
C:            107194340  49528328  57666012    47%  /mnt/c

ejercicio12a  ejercicio13  ejercicio4b  ejercicio5b  ejercicio8  ejercicio82  ejercicio9
ejercicio12a.c ejercicio13.c ejercicio4b.c ejercicio5b.c ejercicio8_1 ejercicio8_2 ejercicio9.c
ejercicio12a.o ejercicio13.o ejercicio4b.o ejercicio5b.o ejercicio8-1 ejercicio8-2.c ejercicio9.o
ejercicio12b  ejercicio4a  ejercicio5a  ejercicio6   ejercicio8_1.c ejercicio8_2.o Informe
ejercicio12b.c ejercicio4a.c ejercicio5a.c ejercicio6.c ejercicio8_1.o ejercicio82.o Makefile
ejercicio12b.o ejercicio4a.o ejercicio5a.o ejercicio6.o ejercicio8-1.o ejercicio8.o

AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/soper-p1$ ./ejercicio8_1 ls df -v
S.ficheros    bloques de 1K  Usados  Disponibles  Uso%  Montado en
rootfs        107194340  49528328  57666012    47%  /
root          107194340  49528328  57666012    47%  /root
home          107194340  49528328  57666012    47%  /home
data          107194340  49528328  57666012    47%  /data
cache         107194340  49528328  57666012    47%  /cache
mnt           107194340  49528328  57666012    47%  /mnt
none          107194340  49528328  57666012    47%  /dev
none          107194340  49528328  57666012    47%  /run
none          107194340  49528328  57666012    47%  /run/lock
none          107194340  49528328  57666012    47%  /run/shm
none          107194340  49528328  57666012    47%  /run/user
C:            107194340  49528328  57666012    47%  /mnt/c

ejercicio12a  ejercicio13  ejercicio4b  ejercicio5b  ejercicio8  ejercicio82  ejercicio9
ejercicio12a.c ejercicio13.c ejercicio4b.c ejercicio5b.c ejercicio8_1 ejercicio8_2 ejercicio9.c
ejercicio12a.o ejercicio13.o ejercicio4b.o ejercicio5b.o ejercicio8-1 ejercicio8-2.c ejercicio9.o
ejercicio12b  ejercicio4a  ejercicio5a  ejercicio6   ejercicio8_1.c ejercicio8_2.o Informe
ejercicio12b.c ejercicio4a.c ejercicio5a.c ejercicio6.c ejercicio8_1.o ejercicio82.o Makefile
ejercicio12b.o ejercicio4a.o ejercicio5a.o ejercicio6.o ejercicio8-1.o ejercicio8.o

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
ejercicio12a  ejercicio13  ejercicio4b  ejercicio5b  ejercicio8  ejercicio82  ejercicio9
ejercicio12a.c ejercicio13.c ejercicio4b.c ejercicio5b.c ejercicio8_1 ejercicio8_2 ejercicio9.c
ejercicio12a.o ejercicio13.o ejercicio4b.o ejercicio5b.o ejercicio8-1 ejercicio8-2.c ejercicio9.o
ejercicio12b  ejercicio4a  ejercicio5a  ejercicio6   ejercicio8_1.c ejercicio8_2.o Informe
ejercicio12b.c ejercicio4a.c ejercicio5a.c ejercicio6.c ejercicio8_1.o ejercicio82.o Makefile
ejercicio12b.o ejercicio4a.o ejercicio5a.o ejercicio6.o ejercicio8-1.o ejercicio8.o

S.ficheros    bloques de 1K  Usados  Disponibles  Uso%  Montado en
rootfs        107194340  49528476  57665864    47%  /
root          107194340  49528476  57665864    47%  /root
home          107194340  49528476  57665864    47%  /home
data          107194340  49528476  57665864    47%  /data
cache         107194340  49528476  57665864    47%  /cache
mnt           107194340  49528476  57665864    47%  /mnt
none          107194340  49528476  57665864    47%  /dev
none          107194340  49528476  57665864    47%  /run
none          107194340  49528476  57665864    47%  /run/lock
none          107194340  49528476  57665864    47%  /run/shm
none          107194340  49528476  57665864    47%  /run/user
C:            107194340  49528476  57665864    47%  /mnt/c

AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/soper-p1$ ./ejercicio8_2 ls df -vp
ejercicio12a  ejercicio13  ejercicio4b  ejercicio5b  ejercicio8  ejercicio82  ejercicio9
ejercicio12a.c ejercicio13.c ejercicio4b.c ejercicio5b.c ejercicio8_1 ejercicio8_2 ejercicio9.c
ejercicio12a.o ejercicio13.o ejercicio4b.o ejercicio5b.o ejercicio8-1 ejercicio8-2.c ejercicio9.o
ejercicio12b  ejercicio4a  ejercicio5a  ejercicio6   ejercicio8_1.c ejercicio8_2.o Informe
ejercicio12b.c ejercicio4a.c ejercicio5a.c ejercicio6.c ejercicio8_1.o ejercicio82.o Makefile
ejercicio12b.o ejercicio4a.o ejercicio5a.o ejercicio6.o ejercicio8-1.o ejercicio8.o

S.ficheros    bloques de 1K  Usados  Disponibles  Uso%  Montado en
rootfs        107194340  49528476  57665864    47%  /
root          107194340  49528476  57665864    47%  /root
home          107194340  49528476  57665864    47%  /home
data          107194340  49528476  57665864    47%  /data
cache         107194340  49528476  57665864    47%  /cache
mnt           107194340  49528476  57665864    47%  /mnt
none          107194340  49528476  57665864    47%  /dev
none          107194340  49528476  57665864    47%  /run
none          107194340  49528476  57665864    47%  /run/lock
none          107194340  49528476  57665864    47%  /run/shm
none          107194340  49528476  57665864    47%  /run/user
C:            107194340  49528476  57665864    47%  /mnt/c

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.

Vamos a explicar el programa. Lo primero que llama la atención es la declaración `fd[2*N_CHILDS][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.

Es importante hacer una observación y es que hay que tener cuidado con el overflow que podría ocasionar la introducción de operandos grandes, ya que el factorial crece bastante rápido. El factorial de 20 ya es igual a 2432902008176640000. Se han utilizado variables de tipo `long long` para hacer el cálculo, pero ya vemos que con un operando grande el cálculo con factoriales y/o potencias desborda enseguida. De seguido, mostramos tres ejecuciones del programa con operandos diferentes.

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/soper-pi$ ./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-pi$ ./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-pi$ ./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-pi$
```

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

Este ejercicio se divide en dos programas (ejercicio12a y ejercicio12b), que hacen la misma función: calcular los N primeros primos. En el primero se calculan los N primeros primos en 100 procesos diferentes simultáneamente y en el segundo, lo mismo, pero esta vez utilizando 100 hilos.

Este ejercicio se enfoca en calcular el tiempo de ambos programas y compararlos, contrastando el resultado obtenido con la teoría.

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.

Analizamos ahora el programa 12a. Comenzamos creando 100 procesos hijos de un único padre, el proceso que lanza el programa. Todos los hijos realizan el cálculo de los 10000 primeros, y el padre los esperará a todos. Se calcula el tiempo entre antes de llamar a los procesos hijos y después de esperarlos a todos para luego mostrarlo por pantalla.

Comentemos ahora un poco el programa 12b. 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 argumentos la dirección de un hilo o `pthread_t`, de segundo los atributos de la creación del hilo, 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 el parámetro de entrada de 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/codigo$ ./ejercicio12a 10000
El tiempo necesitado para crear 100 procesos y calcular los
10000 primes primes en cada uno de ellos ha sido de:
969587300.0 nanosegundos = 0.9696 segundos
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/codigo$ ./ejercicio12b 10000
El tiempo necesitado para crear 100 HILOS y calcular los
10000 primos primos en cada uno de ellos ha sido de:
541367700.0 nanosegundos = 0.5414 segundos
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/codigo$

```

En este ejemplo en particular, el tiempo utilizando procesos ha sido de 0.97 segundos y el mismo programa utilizando hilos ha sido de 0.54 segundos.

El tiempo utilizando hilos ha sido prácticamente la mitad, 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.

10 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 dimensión 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.

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 $n = \text{dim}^2$ 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.

11 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 interacciones 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.