

Sistemas Operativos

Práctica 1

Blanca Martín · Fernando Villar
Grupo 2201 · Pareja 10

03/03/2017

Índice

1. Introducción	1
2. Listado de código fuente	1
3. Ejercicio 4 (a y b)	1
4. Ejercicio 5 (a y b)	4
5. Ejercicio 6	4
6. Ejercicio 8	5
7. Ejercicio 9	5

1. INTRODUCCIÓN

En este documento se discutirán los ejercicios propuestos en la primera práctica de Sistemas Operativos (grupo 2201), que orbitan alrededor de los conceptos relacionados con procesos padre y procesos hijo (a través de funciones como **fork()** y **wait()**, con diferentes análisis e implementaciones para entenderlos en su plenitud; con las familias de funciones **exec**, las diferentes variaciones, utilidades y sintaxis; y con las tuberías, mecanismos de comunicación entre procesos padre y procesos hijo.

2. LISTADO DE CÓDIGO FUENTE

- ejercicio4a.c
- ejercicio4b.c
- ejercicio5a.c
- ejercicio5b.c
- ejercicio6.c
- ejercicio8.c
- ejercicio9.c

Todo ello regulado por el archivo “Makefile”, que genera automáticamente los archivos .o y los ejecutables para cada ejercicio, y que cuenta con la posibilidad de efectuar un comando **make clean** para eliminar todos los archivos generados. La documentación completa del código fuente se presenta en el archivo PDF aportado en este mismo directorio, “Documentación código fuente (Doxygen).pdf”.

3. EJERCICIO 4 (A Y B)

En el código proporcionado en la práctica, se realiza un bucle de tres iteraciones; dentro de este bucle, se genera un proceso hijo mediante **fork()**, y a continuación, el hijo imprime el índice del bucle, así como el padre, que hace exactamente lo mismo. Esto hace que por cada proceso actual se genere un hijo, creando un árbol de procesos en el que en cada iteración, se crea un nodo nuevo por cada nodo ya existente. En concreto, tras tres iteraciones, el proceso inicial tiene tres hijos; el primer hijo del proceso inicial tiene dos; el segundo hijo del proceso inicial tiene uno, y el tercero ninguno; el primer hijo del primer hijo del proceso inicial tiene un hijo, mientras que el segundo ninguno; y el hijo del segundo hijo del proceso inicial tiene otro. Siendo n cada iteración, P_n el número de procesos existentes tras la iteración n , tendríamos que $P_n = 2P_{n-1}$ con $P_0 = 1$, y por tanto $P_n = 2^n$, en nuestro caso $P_3 = 8$.

La diferencia entre los dos códigos es que en el segundo se realiza un solo **wait** por cada proceso. Eso quiere decir que en el código de la segunda versión, como mínimo 4 procesos no van a quedarse huérfanos (puesto que de los 8 procesos que tenemos, 4 no tienen hijos pero 4 sí los tienen, haciendo que cada uno de estos últimos esperen exactamente a uno de sus procesos hijos), pero el resto tienen el riesgo de quedarse, dependiendo de cómo actúe el planificador a la hora de ejecutarlos; sin embargo, en la primera versión el código no contiene ningún **wait**, por lo que pueden quedarse huérfanos todos los procesos o ninguno dependiendo de nuevo del orden de ejecución. En cualquiera de los casos, todo hijo que acabe su ejecución y no haya un padre esperándolo, se queda en estado zombie.

En concreto, en nuestra implementación, hemos añadido una serie de impresiones por pantalla que nos facilitan el análisis de lo que está sucediendo. Por ejemplo, en las siguientes dos imágenes:

```
sliezzan@debian:~$ ps -ef | grep -v grep | sort -n -k 2 | head -n 100
sliezzan@debian:~/Documents/OperatingSystems/S0PER/01_P1$ ./ejercicio4a
PADRE 0, PID: 10749, PPID: 1505
PADRE 1, PID: 10749, PPID: 1505
HIJO 0, PID: 10750, PPID: 10749
PADRE 2, PID: 10749, PPID: 1505
PADRE 1, PID: 10750, PPID: 10749
HIJO 2, PID: 10752, PPID: 10749
HIJO 1, PID: 10751, PPID: 10749
PADRE 2, PID: 10750, PPID: 10749
AUXPROC, PID: 10754, PPID: 10749, RAIZ: 1505
PADRE 2, PID: 10751, PPID: 10749
AUXPROC, PID: 10757, PPID: 10750, RAIZ: 1505
HIJO 2, PID: 10755, PPID: 10750
AUXPROC, PID: 10756, PPID: 10752, RAIZ: 1505
HIJO 2, PID: 10758, PPID: 10751
AUXPROC, PID: 10760, PPID: 10758, RAIZ: 1505
AUXPROC, PID: 10759, PPID: 10755, RAIZ: 1505
HIJO 1, PID: 10753, PPID: 10750
PADRE 2, PID: 10753, PPID: 10750
AUXPROC, PID: 10761, PPID: 10751, RAIZ: 1505
HIJO 2, PID: 10762, PPID: 10753
AUXPROC, PID: 10763, PPID: 10753, RAIZ: 1505
AUXPROC, PID: 10764, PPID: 10762, RAIZ: 1505
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10753)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10755)---pstree(10759)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10753)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10753)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10751)---ejercicio4a(10758)---pstree(10760)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10752)---pstree(10756)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---pstree(10754)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---ejercicio4a(10755)---pstree(10759)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10750)---pstree(10757)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10751)---ejercicio4a(10758)---pstree(10760)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10752)---pstree(10756)
bash(1505)---ejercicio4a(10749)---pstree(10754)
bash(1505)---ejercicio4a(10749)---pstree(10757)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10751)---ejercicio4a(10758)---pstree(10760)
bash(1505)---ejercicio4a(10749)---ejercicio4a(10752)---pstree(10756)
```

Figura 1: Ejecución de ejemplo (1), ejercicio 4a

```
Terminado pstree para 10749.
Terminado pstree para 10750.
Terminado pstree para 10752.
sliezzan@debian-sliezzan:~/Documents/OperatingSystems/SOPER/01_P1$ bash(1505)
bash(1505)
Terminado pstree para 10758.
Terminado pstree para 10755.
bash(1505)
Terminado pstree para 10751.
bash(1505)
bash(1505)
Terminado pstree para 10762.
Terminado pstree para 10753.
```

Figura 2: Ejecución de ejemplo (2), ejercicio 4a

Se puede ver cómo los últimos árboles de procesos indican que dichos procesos se han quedado huérfanos, puesto que lo que se nos muestra es que el proceso raíz (el padre del primer proceso de todos) se encuentra vacío en todo momento, indicando que alguno de los procesos intermedios se ha terminado y sus hijos han pasado a ser propiedad del proceso 1.

En la segunda versión, como mencionamos anteriormente, mínimo 4 procesos terminan de forma correcta, como podemos ver en las siguientes imágenes:

```

slezzan@debian-slezzan:~/Documents/OperatingSystems/SOPER/01_P1$ ./ejercicio4b
PADRE 0, PID: 10593, PPID: 1505
HIJO 0, PID: 10594, PPID: 10593
PADRE 1, PID: 10594, PPID: 10593
PADRE 1, PID: 10593, PPID: 1505
PADRE 2, PID: 10594, PPID: 10593
HIJO 1, PID: 10596, PPID: 10594
PADRE 2, PID: 10593, PPID: 1505
HIJO 2, PID: 10598, PPID: 10593
PADRE 2, PID: 10596, PPID: 10594
PID 10598 no tiene hijos.
HIJO 2, PID: 10599, PPID: 10596
HIJO 1, PID: 10595, PPID: 10593
PID 10599 no tiene hijos.
PADRE 2, PID: 10595, PPID: 10593
AUXPROC, PID: 10600, PPID: 10598, GPID: 1505
HIJO 2, PID: 10597, PPID: 10594
PID 10597 no tiene hijos.
AUXPROC, PID: 10602, PPID: 10599, GPID: 1505
HIJO 2, PID: 10601, PPID: 10595
AUXPROC, PID: 10603, PPID: 10597, GPID: 1505
PID 10601 no tiene hijos.
AUXPROC, PID: 10604, PPID: 10601, GPID: 1505
bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)—ejercicio4b(10596)—ejercicio4b(10599)—pstree(10602)
                                     |
                                     |—ejercicio4b(10597)—pstree(10603)
                                     |
                                     |—ejercicio4b(10595)—ejercicio4b(10601)—pstree(10604)
                                     |
                                     |—ejercicio4b(10598)—pstree(10600)

Terminado pstree para 10597.
Terminado proceso 10597.
AUXPROC, PID: 10605, PPID: 10594, GPID: 1505
bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)—ejercicio4b(10596)—ejercicio4b(10599)—pstree(10602)
                                     |
                                     |—ejercicio4b(10595)—ejercicio4b(10601)—pstree(10604)
                                     |
                                     |—ejercicio4b(10598)—pstree(10600)

Terminado pstree para 10599.
Terminado proceso 10599.
AUXPROC, PID: 10606, PPID: 10596, GPID: 1505
bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)—ejercicio4b(10596)
                                     |
                                     |—ejercicio4b(10595)—ejercicio4b(10601)
                                     |
                                     |—ejercicio4b(10598)—pstree(10600)

Terminado pstree para 10598.
Terminado proceso 10598.

```

Figura 3: Ejecución de ejemplo (1), ejercicio 4b

```

bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)—ejercicio4b(10596)
                                     |
                                     |—ejercicio4b(10595)—ejercicio4b(10601)—pstree(10604)
                                     |
                                     |—ejercicio4b(10598)—pstree(10600)

Terminado pstree para 10601.
bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)—ejercicio4b(10596)
                                     |
                                     |—pstree(10605)
                                     |
                                     |—ejercicio4b(10595)—ejercicio4b(10601)—pstree(10604)
                                     |
                                     |—ejercicio4b(10598)

Terminado proceso 10601.
AUXPROC, PID: 10607, PPID: 10593, GPID: 1505
Terminado pstree para 10594.
AUXPROC, PID: 10608, PPID: 10595, GPID: 1505
bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)
                                     |
                                     |—ejercicio4b(10595)

Terminado pstree para 10596.
bash(1505)—ejercicio4b(10593)—ejercicio4b(10594)
                                     |
                                     |—ejercicio4b(10595)—pstree(10608)
                                     |
                                     |—pstree(10607)

Terminado pstree para 10593.
bash(1505)
Terminado pstree para 10595.

```

Figura 4: Ejecución de ejemplo (2), ejercicio 4b

En efecto, las cuatro hojas del árbol de procesos finalizan correctamente, mientras que para los demás ocurre que uno se ha quedado huérfano, como podemos ver por el árbol del proceso raíz del final, que se encuentra vacío (probablemente porque el proceso padre ha finalizado antes que el último de sus hijos restantes).

4. EJERCICIO 5 (A Y B)

■ Caso A:

Para que se generen un conjunto de procesos de forma secuencial, tenemos que asegurarnos de que cada vez que se crea un hijo, el padre no vuelve a crear ningún hijo más. Por tanto, la implementación con menos diferencias respecto al ejercicio anterior para realizar esta tarea, es colocar un **break** en el código del padre, de manera que cada vez que un proceso genere un hijo, el hijo pueda seguir iterando en el bucle, pero el padre no lo haga y por tanto no cree ningún hijo más. Para asegurarnos de que ningún proceso queda huérfano, añadimos un **wait(NULL)** al final del código, de manera que ningún padre termine sin antes haber esperado a que la ejecución del código de su único hijo termine.

■ Caso B:

Para que se generen un conjunto de procesos en paralelo, tenemos que asegurarnos de que cada vez que se crea un hijo, este último no vuelve a crear ningún hijo más. Por tanto, la implementación con menos diferencias respecto al ejercicio anterior para realizar esta tarea, es colocar un **break** en el código del hijo, de manera que cada vez que un proceso genere un hijo, el padre pueda seguir iterando en el bucle, pero el hijo no lo haga y por tanto no cree ningún hijo más. Para asegurarnos de que ningún proceso queda huérfano, añadimos un **wait(NULL)** en el código del padre dentro del bucle, de manera que en cada iteración espere a que el hijo que ha generado termine su ejecución, consiguiendo por tanto que no queden procesos huérfanos en el camino.

5. EJERCICIO 6

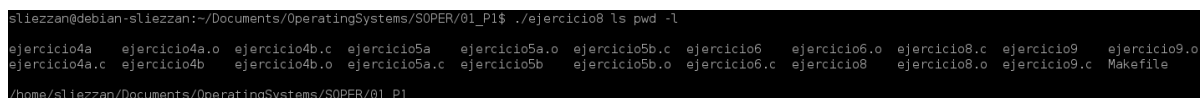
En este ejercicio, si en el proceso hijo se pide al usuario que introduzca una cadena para guardarla, el padre no tiene acceso a esa información; esto es debido a que en la generación del hijo se hace una copia de la información contenida en el padre, y desde ese momento sólo el hijo puede acceder a ella. Finalmente, la memoria hay que liberarla en ambos procesos, puesto que el sistema exporta también el hecho de que se trata de memoria dinámica. Todo esto se puede ver en la siguiente imagen:

```
sliezzan@debian-sliezzan:~/Documents/OperatingSystems/SOPER/01_P1$ valgrind --tool=memcheck ./ejercicio6
==14320== Memcheck, a memory error detector
==14320== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==14320== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==14320== Command: ./ejercicio6
==14320==
PADRE 14320
HIJO 14321
Introduce un nombre: as
El nombre introducido para 14321 es as
==14321==
==14321== HEAP SUMMARY:
==14321==    in use at exit: 0 bytes in 0 blocks
==14321==   total heap usage: 1 allocs, 1 frees, 80 bytes allocated
==14321==
==14321== All heap blocks were freed -- no leaks are possible
==14321==
==14321== For counts of detected and suppressed errors, rerun with: -v
==14321== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==14320== Conditional jump or move depends on uninitialised value(s)
==14320==    at 0x4E7FDCC: vfprintf (vfprintf.c:1642)
==14320==    by 0x4E85CE6: fprintf (fprintf.c:32)
==14320==    by 0x400924: main (in /home/sliezzan/Documents/OperatingSystems/SOPER/01_P1/ejercicio6)
==14320==
El nombre introducido para 14320 es
==14320==
==14320== HEAP SUMMARY:
==14320==    in use at exit: 0 bytes in 0 blocks
==14320==   total heap usage: 1 allocs, 1 frees, 80 bytes allocated
==14320==
==14320== All heap blocks were freed -- no leaks are possible
==14320==
==14320== For counts of detected and suppressed errors, rerun with: -v
==14320== Use --track-origins=yes to see where uninitialised values come from
==14320== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Figura 5: Ejecución de ejemplo, ejercicio 6

6. EJERCICIO 8

Para la realización de este ejercicio se ha asumido que los ejecutables de los comandos se encuentran o bien en `/usr/bin` o bien en `/bin`, si no se encontraran en ninguno de estos dos directorios, se imprimirá un mensaje por pantalla indicando que ha habido este error (también se puede imprimir si la instrucción es desconocida). Por lo demás, tal y como se da a entender en el enunciado, se ha asumido que cada comando es singular (es decir, no lleva etiquetas ni palabras adicionales), y por tanto el método ha sido desarrollado alrededor de un array de dos strings, el primero con el comando y el segundo llevado a NULL, para que valga en aquellas funciones `exec` que requieren un array de strings como argumento. La ejecución se puede ver en la siguiente imagen:



```
sliezzan@debian-sliezzan:~/Documents/OperatingSystems/SOPER/01_P1$ ./ejercicio8 ls pwd -l
ejercicio4a.o  ejercicio4a.o  ejercicio4b.c  ejercicio5a    ejercicio5a.o  ejercicio5b.c  ejercicio6    ejercicio6.o  ejercicio8.c  ejercicio9    ejercicio9.o
ejercicio4a.c  ejercicio4b    ejercicio4b.o  ejercicio5a.c  ejercicio5b    ejercicio5b.o  ejercicio6.c  ejercicio8    ejercicio8.o  ejercicio9.c  Makefile
/home/sliezzan/Documents/OperatingSystems/SOPER/01_P1
```

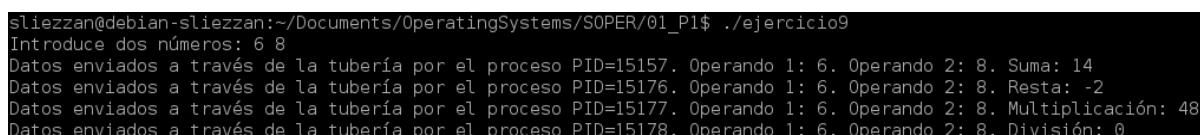
Figura 6: Ejecución de ejemplo, ejercicio 8

7. EJERCICIO 9

En este problema, hemos optado por crear dos tuberías (una con comunicación padre - hijo y otra con comunicación hijo - padre) de manera que el proceso padre lee los números por teclado y los manda al hijo; en el hijo, se decide qué hacer según la iteración del bucle principal (que marca la operación a realizar según:

- $i = 0$: Suma
- $i = 1$: Resta
- $i = 2$: Multiplicación
- $i = 3$: División

a través del índice i del bucle), y se manda la cadena correspondiente al padre, que la imprime y realiza un `wait` para evitar que el hijo quede huérfano o zombie. La imagen siguiente describe el proceso:



```
sliezzan@debian-sliezzan:~/Documents/OperatingSystems/SOPER/01_P1$ ./ejercicio9
Introduce dos números: 6 8
Datos enviados a través de la tubería por el proceso PID=15157. Operando 1: 6. Operando 2: 8. Suma: 14
Datos enviados a través de la tubería por el proceso PID=15176. Operando 1: 6. Operando 2: 8. Resta: -2
Datos enviados a través de la tubería por el proceso PID=15177. Operando 1: 6. Operando 2: 8. Multiplicación: 48
Datos enviados a través de la tubería por el proceso PID=15178. Operando 1: 6. Operando 2: 8. División: 0
```

Figura 7: Ejecución de ejemplo, ejercicio 9