

# INFORME PRÁCTICA 2

Alejandro Santorum Varela - alejandro.santorum@estudiante.uam.es

David Cabornero Pascual - david.cabornero@estudiante.uam.es

Pareja 7

Universidad Autónoma de Madrid

05-04-2018

## Contents

1	Introducción	2
2	Ejercicio 2	2
3	Ejercicio 4	2
4	Ejercicio 6-A	4
5	Ejercicio 6-B	5
6	Biblioteca de Semaforos	6
7	Ejercicio 9	6
8	Conclusión y comentarios finales	11

# 1 Introducción

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

Para cada ejercicio se comentará su diseño, su funcionalidad detalla y/o 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 él en la carpeta de la entrega.

## 2 Ejercicio 2

Empezamos con el primer ejercicio entregable de la práctica, el ejercicio 2.

El enunciado del ejercicio era claro: un proceso padre crea cuatro procesos hijos, los cuales muestran por pantalla “Soy el proceso hijo [PID]” para después dormirse durante 30 segundos. Mientras tanto, el proceso padre duerme 5 segundos y a continuación envía la señal **SIGTERM** mediante la función `kill`. Después de dormirse durante 30 segundos, los procesos hijos deberían mostrar por pantalla “Soy el proceso hijo [PID] y ya me toca terminar” y finalizar, pero esto no sucede porque antes de acabar esos 30 segundos el padre les envía la dicha señal, pero provoca que finalicen antes de tiempo, por lo que nunca van a imprimir lo segundo.

Abajo podemos comprobar la salida de este sencillo programa, y vemos que contrasta con los analizado anteriormente:

```
santorum:~/workspace/Practice2 (master) $ ./ejercicio2
Soy el proceso hijo 1316
Soy el proceso hijo 1317
Soy el proceso hijo 1318
Soy el proceso hijo 1319
santorum:~/workspace/Practice2 (master) $ █
```

## 3 Ejercicio 4

Continuamos comentando el ejercicio 4, el segundo ejercicio a realizar. Este ya tiene un poco de complejidad ya que entran nuevas funciones con las cuales nunca hemos trabajado.

Su funcionalidad consiste en que el proceso padre crea un proceso hijo que realizará un trabajo (imprimir “Soy [PID] y estoy trabajando”). Pasado un tiempo, solicitará al padre que le releve un nuevo proceso hijo. El padre creará un nuevo proceso hijo y será, este nuevo proceso hijo, el encargado de avisar al hijo anterior de que ya puede terminar. Una vez avisado, el nuevo hijo empezará a trabajar de la misma manera.

Utilizaremos dos tipos de señales, **SIGTERM**, que será utilizada por cada proceso hijo para avisar al anterior hijo para que finalice (menos el último proceso hijo que será avisado

por el padre). Además, también necesitaremos la señal **SIGUSR1** a la que le cambiaremos el manejador por defecto para que simplemente no ignore la señal y despertar al proceso padre de la función `pause()`.

Mostramos a continuación la salida de este ejercicio con número de procesos hijos igual a tres:

```
santorum:~/workspace/Practice2 (master) $ ./ejercicio4 3
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando
Soy 1407 y estoy trabajando despues de enviar la señal
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando
Soy 1408 y estoy trabajando despues de enviar la señal
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando
Soy 1410 y estoy trabajando despues de enviar la señal
santorum:~/workspace/Practice2 (master) $ █
```

Poco que comentar esta salida. Los procesos hijos realizan su trabajo: imprimir 10 veces ese mensaje, y después envía la señal al padre para que lo releven.

Hemos añadido nosotros por nuestra cuenta una instrucción que consiste en mostrar por pantalla "Soy [PID] y estoy trabajando despues de enviar la señal" para informarnos de cuanto tiene que esperar el proceso hijo desde que envía la señal al padre, hasta que el nuevo proceso hijo le envía la señal de finalización. Vemos que solo tiene tiempo a imprimir una instrucción.

Comentar que el enunciado no aclara un detalle: pone que "Pasado un tiempo de 10 segundos (tras imprimir 10 veces el mensaje)..." lo cual no queda claro si hay que esperar adicionalmente 10 segundos después de todas las impresiones, o por el contrario, ya cuenta el tiempo usado para mostrar por pantalla los mensajes.

Hemos optado por usar la primera variante, esperar 10 segundos **después de todas** las impresiones.

## 4 Ejercicio 6-A

Avanzamos hasta el siguiente ejercicio entregable, el ejercicio 6a. Se pide modificar el código proporcionado para que un proceso hijo, después de ser creado, establezca una alarma de 40 segundos. A continuación comenzará a ejecutar un bucle indefinidamente haciendo principalmente tres acciones: bloquear las señales **SIGALRM**, **SIGUSR1** y **SIGUSR2**, imprimir por pantalla los números 0, 1, ..., 4, y por último, desbloquear las señales **SIGALRM** y **SIGUSR1**, durmiendo antes de comenzar el bucle de nuevo durante 3 segundos.

El objetivo de bloquear las señales y después desbloquearlas es simple: impedir que finalice la ejecución del proceso cuando está realizando el trabajo, la impresión de números. Entonces, si la señal de alarma llega cuando está bloqueada **SIGALRM**, está esperando a que se desbloquee y el proceso finalizará normalmente.

La salida de este ejercicio es la que se muestra a continuación:

```
santorum:~/workspace/Practice2 (master) $ ./ejercicio6a
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
santorum:~/workspace/Practice2 (master) $ █
```

Como la alarma está programada para 40 segundos, y el proceso hijo imprime 5 números en un intervalo de 5 segundos más una espera de tres, el proceso tiene tiempo para imprimir 5 veces el conjunto de números, tal y como se muestra en la salida.

Como se ha comentado anteriormente, si el proceso hijo recibiese la señal de alarma cuando esta está bloqueada, esta pasaría a estar en espera a que este desbloqueada y el proceso responda a ella, por lo que nunca el proceso va a acabar en mitad del trabajo de impresión.

## 5 Ejercicio 6-B

Ahora explicamos la segunda parte del ejercicio 6, el apartado b. En este apartado el proceso principal (padre) crea un proceso hijo que realiza el mismo trabajo que el que realizaba en el apartado a. El padre esperará 40 segundos para a continuación enviar la señal **SIGTERM** para que el proceso hijo imprima "Soy [PID] y he recibido la señal SIGTERM" y finalice. Este procedimiento lo implementamos en un nuevo manejador distinto al de por defecto de la señal SIGTERM, por lo que necesitamos la ayuda de la función `signal(...)`. Abajo mostramos la salida del programa comentado:

```
santorum:~/workspace/Practice2 (master) $ ./ejercicio6b
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
Soy 1708 y he recibido la señal SIGTERM
santorum:~/workspace/Practice2 (master) $
```

Podemos ver que el proceso hijo ha hecho la función requerida, y cuando han pasado 40 segundos el proceso padre le envía la señal para que actúe como requería el enunciado.

Es importante comentar que cuando el proceso padre acaba la espera de 40 segundos y envía la señal de finalización al hijo, este justo acaba de imprimir el último número del turno. No obstante, a diferencia del ejercicio 6a, si la recibiese en otro momento, **inter-rupiría su ejecución y antendería la señal** ya que esta **es la única que no está bloqueada**.

Modificando el tiempo de espera podemos apreciar esto. Aquí mostramos la salida solo cambiando el valor de la función `sleep()`:

```
santorum:~/workspace/Practice2 (master) $ ./ejercicio6b
0
1
2
3
4
0
1
Soy 1732 y he recibido la señal SIGTERM
santorum:~/workspace/Practice2 (master) $
```

## 6 Biblioteca de Semaforos

Llegamos al ejercicio donde tenemos que implementar la biblioteca de semaforos. Esta biblioteca, aparte de un gran ejercicio didáctico, si la logramos implementar correctamente nos servirá (ella o una pequeña modificación) enormemente en el futuro.

En primer lugar necesitamos indicar que las **cabeceras de las funciones aportadas por la documentación de la práctica no siguen la guía de estilo estándar de C** exigidas para las prácticas. Hemos hecho las funciones tal y como se indican en sus cabeceras, pero queremos dejar claro que tenemos la guía de estilo en cuenta.

Por otro lado, nos gustaría comentar un error que creemos (casi con seguridad) que tiene la especificación de las funciones `UpMultiple_Semaforo(...)` y `DownMultiple_Semaforo(...)`. Se trata del parámetro de entrada `size`, que la documentación indica que es el tamaño del array de semáforos, pero nosotros lo hemos **adoptado como el tamaño del array de enteros \*active**, ya que el tamaño del array de semaforos no es importante para `semop()`, pero en cambio el tamaño del array de enteros sí que es muy importante, tanto para la inicialización de la estructura `sembuf`, como para la funcion `semop`. Creemos que es una errata de la documentación.

Subrayar que las funciones, cuando devuelven `ERROR`, también imprimen por pantalla un mensaje explicativo de cual pudo haber sido el error. Esto es de gran utilidad para *debuggear* en casos de fallo.

Aparte de esto, tenemos poco que comentar en cuanto a este ejercicio ya que no tenemos una forma rápida de comprobar si está bien implementado y la implementación se basa en estudiar el man de linux junto con los ejemplos de la documentación. Para ello utilizaremos el próximo ejercicio, que es bastante completo.

## 7 Ejercicio 9

Llegamos al último ejercicio de la práctica 2. No se puede decir que haya sido *coser y cantar*, de hecho, el apartado de decisiones de diseño, dificultades encontradas y posibles futuras mejores es de un tamaño bastante importante.

Lo primero de todo, vamos a comentar poco a poco el fichero fuente 'ejercicio9.c' debido a que no tiene una envergadura adecuada para ser analizado si uno no ha participado en el desarrollo del mismo. Para ello se han incluido grandes cantidades de comentarios (en contra de la guía de estilo estándar de C, pero a favor de las peticiones de las documentaciones de las prácticas).

En primer lugar ya sabemos lo que toca, comprobación de errores de entrada: el número

de argumentos tiene que ser igual a dos, el nombre y el número de cajeros del programa. Además, se incluye una función auxiliar implementada por nosotros `'is_valid_integer(char*)'` que tiene el objetivo de comprobar que realmente el segundo argumento es un entero.

Lo siguiente que hacemos es crear e inicializar los semáforos del programa: creamos un array de  $n + 1$  semáforos, siendo  $n$  el número de cajeros. Para esto nos valemos de las funciones de la biblioteca de semáforos. Subrayar una decisión de código: la **key** para la creación de semáforos (la cual será proporcionada a la función `ftok(PATH, KEY)`), es escogida aleatoriamente entre  $MIN\_KEY = 100$  y  $MAX\_KEY = 99999$  para reducir posibles errores (quizá el semáforo ya está creado, lo cual es mejor evitar) y hacer el sistema más robusto y portable a diferentes SSOO.

Utilizaremos los semáforos de índices  $(1, 2, \dots, n)$  para los ficheros que simularán las cajas de los cajeros y el semáforo de índice igual a *cero* será el encargado de proteger el fichero de texto auxiliar para la comunicación de los procesos hijos con el padre. Todos los semáforos se inicializan a uno.

A continuación, creamos una máscara de señales con todas menos SIGUSR1 (utilizada por los hijos, cajeros, para avisar al padre de que retire dinero de su caja) y SIGINT, por si necesitamos utilizar `ctrl+C`. Esta máscara es activada de inmediato, siendo heredada en el futuro por los procesos hijos, y dejará de tener efecto al finalizar el programa (el S.O. se encargará de eso por nosotros).

Es en este momento cuando creamos  $n = \text{numero de cajeros} = \text{numero de ficheros de texto de clientes}$ . Cada fichero de clientes le corresponderá a un cajero y tendrá una longitud de 50 transacciones. El coste de las transacciones es determinando aleatoriamente entre 0 y 300 tal y como indica en la documentación.

Ahora se crea el fichero de comunicación entre cajeros y gerente (hijos y padre), se asigna el nuevo manejador de SIGUSR1 y se crean los  $n$  procesos hijos que simularán los cajeros.

A partir de este momento es cuando empieza "lo divertido" o mejor dicho, lo complicado. Por un lado, los procesos hijos usarán las funciones **`nombre_transaccion(int i)`** y **`nombre_cajero(int i)`**, que reciben el id (no el PID, el id del cajero) y devuelven el fichero de clientes (compras) y el fichero donde llevarán la contabilidad de la caja respectivamente. Gracias a ellas agilizarán la obtención de los nombres de archivos.

De seguido, cada cajero leerá de su archivo de clientes correspondiente (sin necesidad de semáforo porque nadie, además de ellos, intenta usar ese fichero) el valor de la transacción para a continuación, **bajando su semáforo de fichero**, actualizar la contabilidad de su caja, y **subir de nuevo el semáforo de caja**. Una vez terminando con esta secuencia de instrucciones, el cajero comprobará la cantidad actualizada de su caja, y si esta supera los 1000 euros, **levantará el semáforo del fichero de cola de señales**, para introducir su id junto con una palabra indicadora de la acción a realizar, *filled* para pedir que el padre le quite 900 euros y se los lleve a la cuenta global, o *fin* que indica que el cajero ha terminado de procesar ya las 50 transacciones de clientes. Una vez escrito esto en el fichero de cola, se

enviará una señal SIGUSR1 al padre para despertarlo y el cajero **levantará el semáforo del fichero de la cola** para poder continuar trabajando, no sin antes realizar una espera de tiempo aleatorio entre uno y cinco segundos, que simula el tiempo de atención a un cliente real.

Mientras todo esto pasa, el proceso padre entrará en un bucle mientras el número de cajeros finalizados sea menor que el número de cajeros totales. Dentro de este bucle se pondrá a dormir usando la función **sigsuspend(&set)** esperando a que algunos de los cajeros lo reclame.

Una vez que sea despertado, el proceso padre (gerente) **bajará el semáforo del fichero de cola de señales** para leer el id del cajero que lo llama, junto con la acción a realizar. Somos conscientes que una variante de este método sería que los cajeros enviaran dos señales diferentes (por ejemplo, SIGUSR1 y SIGUSR2), una solicitando que el padre les retire 900 euros (manejador de SIGUSR1) y la otra solicitando que le vacíe la caja porque ha finalizado (manejador de SIGUSR2). No se ha adoptado este método por simplicidad del código, ya que seguir esta línea implicaría crear un manejador nuevo, nuevas variables globales, y una máscara diferente; pero este es la cruz a la que se tiene que someter un programador: memoria vs eficiencia, no siempre las dos son mejorables al mismo tiempo.

Siguiendo con la explicación del funcionamiento, una vez que el proceso padre sepa que tiene que hacer, **bajará el semáforo de la caja correspondiente**, haciendo los cambios pertinentes y volviendo a **levantar el semáforo de la caja** al momento. Una vez terminado con el archivo de cola de señales, el padre lo vaciará para poder saber en el futuro, cuales son las señales no leídas (lógicamente, las que no han sido borradas aún), y finalmente, **levantará el semáforo del semáforo de cola**.

Es importante en este momento explicar el funcionamiento de la única variable global utilizada: **flag**. Su misión es encargarse de informarle al padre **si una señal SIGUSR1 ha llegado antes de que este se ponga de nuevo a dormir**. Tenemos que tener en cuenta que desde que el padre levanta el semáforo de caja, hasta que se pone de nuevo en el estado de *sleeping*, pasa un tiempo insignificante, solo dos comprobaciones lógicas. En cambio, cualquier cajero tendría que abrir un fichero, leerlo, hacer alguna operación, escribir en el fichero, cerrar el fichero y enviarle la señal al padre en el tiempo que este hace dos míseras comprobaciones lógicas.

Esto parece de poca importancia en la práctica, pero en la teoría sabemos que no podemos **hacer suposiciones sobre el tiempo de ejecución de ningún proceso, por inverosímil que parezca**, que ahí que hagamos **sigsuspend(...)** si y solo si flag es igual a cero, lo que quiere decir que no hemos entrado en el manejador de SIGUSR1, que pone flag a 1.

Una vez que todos los cajeros hayan finalizado, el proceso padre **eliminará los semáforos** y hará el recuento total de las cajas, utilizando la función **final\_check(int n, int total)**, que lo que hace es sumas todas las transacciones de los ficheros de clientes y comparar el resultado con el obtenido por el gerente (proceso padre) después de procesar todas las señales.



Antes de dar paso a un ejemplo de salida de este programa, tenemos que hacer un par de aclaraciones y/o comentarios.

En primer lugar, a simple vista parece que el 'ejercicio9' es de un tamaño significativo, pero si le quitamos las líneas de comentarios y las de comprobaciones de errores (que no son pocas precisamente, es lo que tiene trabajar de una forma segura y robusta), nos quedamos con un código "simple". Lo único que se hace es que varios procesos utilicen ficheros de texto, pero esto nos sirve para comprobar el correcto funcionamiento de nuestros semáforos, además de servir de un ejercicio didáctico enorme.

Por último, nos gustaría subrayar una decisión de diseño importante. A la hora de subir o bajar semáforos utilizamos **espera activa**, que es una técnica donde un proceso verifica repetidamente una condición, en este caso lo normal es no REPETIR DICHA VERIFICACIÓN, pero puede darse el caso que no se pueda hacer la bajada/subida de semáforo por cualquier razón del sistema. Esta es la razón por la que se realiza una espera activa. Comentar que nunca hemos abandonado la posibilidad de que realmente suceda un error con los semáforos, por lo que hemos añadido una comprobación de intentos máximos de espera activa (directamente proporcional al número de cajeros). Si se supera el número de intentos máximo, se considera que el semáforo ha fallado por completo, y cortamos la ejecución del programa de inmediato, liberando todos los recursos y comunicándoselo al usuario por pantalla.

Por fin, ahora vamos a mostrar una salida de este programa:

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/P2-ds
==56== Memcheck, a memory error detector
==56== copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==56== Using valgrind-3.14.0.SVN and LibVEX; rerun with -h for copyright info
==56== Command: ./ejercicio9 2
==56==
Soy el cajero 1 con PID=57 y estoy cobrando 52 (total caja = 52)
Soy el cajero 2 con PID=58 y estoy cobrando 141 (total caja = 141)
Soy el cajero 1 con PID=57 y estoy cobrando 173 (total caja = 225)
Soy el cajero 2 con PID=58 y estoy cobrando 72 (total caja = 213)
Soy el cajero 1 con PID=57 y estoy cobrando 127 (total caja = 352)
Soy el cajero 2 con PID=58 y estoy cobrando 72 (total caja = 285)
Soy el cajero 1 con PID=57 y estoy cobrando 124 (total caja = 476)
Soy el cajero 2 con PID=58 y estoy cobrando 13 (total caja = 298)
Soy el cajero 1 con PID=57 y estoy cobrando 89 (total caja = 565)
Soy el cajero 2 con PID=58 y estoy cobrando 70 (total caja = 368)
Soy el cajero 1 con PID=57 y estoy cobrando 210 (total caja = 775)
Soy el cajero 2 con PID=58 y estoy cobrando 60 (total caja = 428)
```

La salida es más amplia que lo que mostramos ya que se ha incluido una cantidad considerable de impresiones por pantalla para darnos información del desarrollo de los cajeros. Otra razón de peso para mostrar una salida así de poblada, es para no inquietarnos esperando a que el programa acabe, ya que la terminal estará parpadeando durante una media de  $\frac{2.5 \cdot 60 \cdot n}{60}$  minutos, que para solo dos procesos son aproximadamente 4 minutos, lo cual hace pensar muchas veces que el programa *se ha colgado*. Si no desea esta salida, puede simplemente comentar 1 línea de código.

Ahora mostramos la parte final de la salida del programa, omitiendo el resto de "Soy el cajero [ID] con PID=[PID] y estoy cobrando [precio\_compra] (total caja = [cantidad\_caja\_actual])".

```
Soy el cajero 2 con PID=58 y estoy cobrando 151 (total caja = 1072)
Soy el cajero 1 con PID=57 y estoy cobrando 105 (total caja = 905)
Soy el cajero 2 con PID=58 y estoy cobrando 58 (total caja = 230)
Soy el cajero 1 con PID=57 y estoy cobrando 213 (total caja = 1118)
Soy el cajero 2 con PID=58 y estoy cobrando 50 (total caja = 280)
Soy la caja 1 con PID=57 y he finalizado con 10 peticiones
Soy la caja 2 con PID=58 y he finalizado con 8 peticiones
```

los cajeros informan al usuario que han finalizado con éxito

```
Comprobando la correspondencia de las ganancias...
==57== Suma ficheros transacciones = 14898
==57== HEAP SUMMARY:
==57==   in use at exit: 0 bytes in 0 blocks
==57==   total heap usage: 338 allocs, 338 frees, 762,796 bytes allocated
==57== Suma total de los cajeros = 14898
==58==
==57== All heap blocks were freed -- no leaks are possible
Comprobado exitosamente: la suma coincide con el total
==58== HEAP SUMMARY:
==58==   in use at exit: 0 bytes in 0 blocks
==58==   total heap usage: 334 allocs, 334 frees, 753,500 bytes allocated
==58==
==57== All heap blocks were freed -- no leaks are possible
==57== For counts of detected and suppressed errors, rerun with: -v
==58== ==> GANANCIAS = 14898 euros <==
==57== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Se han realizado un total de 18 peticiones (comprobar que es la suma de las peticiones de los hijos)
==58== For counts of detected and suppressed errors, rerun with: -v
==58== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==56==
==56== HEAP SUMMARY:
==56==   in use at exit: 0 bytes in 0 blocks
==56==   total heap usage: 182 allocs, 182 frees, 286,996 bytes allocated
==56==
==56== All heap blocks were freed -- no leaks are possible
==56== For counts of detected and suppressed errors, rerun with: -v
==56== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/users/Alejandro Santorum/Desktop/P2-definitivo$
```

La suma de las compras de los clientes es igual al valor calculado por el gerente y por los deferentes cajeros

La suma del número de señales enviadas por los cajeros es igual al número de respuestas hechas por el proc. padre:  
10 + 8 = 18

0 errores de memoria y el resultado obtenido se acerca mucho a la media esperada:  
media = 50\*150\*2 = 15000  
obtenido = 14898

Invitamos a probar diferentes numeros de cajeros, pero siempre teniendo en cuenta el tiempo que puede llegar a tardar la demora:  $t_M = \frac{2.5*60*n}{60}$  ( $t_M = tiempoMedio$ ).

Una vez que se termine de probar/ejecutar el programa las veces que se quiera, recomendamos ejecutar **make clean** para limpiar los archivos binarios compilados, y **make text** para eliminar todos los ficheros de texto de la carpeta `txtfiles`. Estos ficheros, si el programa se ha ejecutado con exito, deberán estar en el siguiente estado: el fichero de cola de señales estará vacío (el proceso padre ha leído todas las señales), los archivos que simulan las cajas (*cajero<sub>i</sub>*) deberán tener escrito el número cero, lo que quiere decir que todo su dinero ha sido retirado satisfactoriamente. Por último, en los ficheros de clientes (*clientesCaja<sub>i</sub>*) estarán las transacciones de los clientes creadas en la última ejecución del programa. Hemos dejado los ficheros de texto sin eliminarlos para hacer cualquier comprobación de error.

## 8 Conclusión y comentarios finales

Señalar que en los primeros cuatro ejercicios no se ha incluido el informe de *Valgrind* pero sabemos con certeza que no tienen ningún error de memoria.

Ahora sí, una vez finalizada la práctica tenemos que echar una mirada atrás y ver todo lo que hemos aprendido. Hemos tomado contacto con las señales entre procesos y con los semáforos. Han sido 2-3 semanas intensas con una gran cantidad de funciones nuevas que jamás habíamos utilizado, y que ahora nos permitirán realizar programas mucho más exigentes.

Nos llevamos un sabor agri dulce, ya que el ejercicio9, el destinado a utilizar semáforos, era ciertamente enrevesando, ya que nos obligaba a comunicar de alguna forma dos procesos al mismo tiempo que utilizábamos semáforos y enviábamos señales. Es una pena que tengamos que realizar ciertos "apaños" rudimentarios para poder coordinar los procesos, pero es lo que tenemos a mano actualmente, hasta que veamos los temas futuros.

Error tras error se ha ido aprendiendo y esperamos haber hecho una buena práctica.