# Memoria de la práctica 2

Rafael Sánchez Sánchez - Sergio Galán Martín
Sistemas Operativos
Universidad Autónoma de Madrid

# 1 Ejercicio 1.

En este ejercicio se nos pide contestar a una seria de preguntas cortas.

## 1.1 Respuestas a preguntas cortas.

**Ejecuta en línea de comando la orden $ kill -l. ¿Qué obtienes?**
Obtenemos la lista de todas la señales del sistema asociadas con su número representativo.

## 1.2 Salida del programa.

Ejercicio 1

```
$ kill -l

 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

# 2 Ejercicio 2.

En este ejercicio se nos pide escribir un programa que siga la especificación dada en la documentación de la práctica y contestar algunas cuestiones acerca de su ejecución.

## 2.1 Respuestas a preguntas cortas.

**¿Qué mensajes imprime cada hijo tras la ejecución del programa especificado en la documentación de la práctica?¿Por qué?**
Vemos que cada hijo imprime solamente el mensaje "Soy el proceso hijo $(pid)", ya que al mandar el padre `SIGTERM` antes de que el hijo haya hecho la espera de 30 segundos, hace que termine antes de llegar al segundo `printf`.

### 2.2 Salida del programa.

Ejercicio 2

```
$ ./build/ejercicio2

Soy el proceso hijo 2736.
Soy el proceso hijo 2737.
Soy el proceso hijo 2738.
Soy el proceso hijo 2739.
```

## 3 Ejercicio 3.

En este ejercicio se nos pide contestar a una seria de preguntas cortas.

### 3.1 Respuestas a preguntas cortas.

**a. ¿La llamada a `signal` hace que se ejecute la función de captura?**
No, simplemente asocia la señal `SIGINT` al manejador captura. Es decir, si se recibe la señal se ejecutará captura y en caso contrario no. Sin embargo, al haber un `while` 1, o le mandamos alguna señal, ya sea `SIGINT` o `SIGKILL`, o nunca acabará el programa.
**b. ¿Cuándo aparece el `printf` en pantalla?**
Cuando se recibe la señal, antes de que termine el programa.
**c. ¿Qué ocurre por defecto cuando un programa recibe una señal y no la tiene capturada?**
Se ejecuta el manejador por defecto del sistema, que puede ser ignorarla, terminar la ejecución, realizar cierta acción...
**d. El código de un programa captura la señal `SIGKILL` y la función manejadora escribe "He conseguido capturar SIGKILL". ¿Por qué nunca sale por pantalla "He conseguido capturar SIGKILL"?**
Porque la señal SIGKILL no es capturable al terminar instantáneamente la ejecución del programa.

## 4 Ejercicio 4.

En este ejercicio se nos pide escribir un programa que siga la especificación dada en la documentación de la práctica.

### 4.1 Decisiones de diseño.

En este ejercicio decidimos usar `pause()` debido a la poca complejidad del mismo y a que a la altura de este ejercicio no se habían introducido las máscaras de señales, por lo que creímos conveniente usar esta función para usar máscaras en ejercicios posteriores. La señal

que reactiva el programa es `SIGUSR1`, para la que hemos definido un manejador (que no hace nada).

Además, hemos definido la función `aredigits(1)` para asegurar que el parámetro de entrada del programa es una cadena de caracteres numéricos.

## 4.2 Salida del programa.

Ejercicio 4

```
$ ./build/ejercicio4 2

Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26033 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
Soy 26040 y estoy trabajando
```

# 5 Ejercicio 6a.

En este ejercicio se nos pide modificar un programa para que siga la especificación dada en la documentación de la práctica y contestar algunas cuestiones acerca de su ejecución.

## 5.1 Respuestas a preguntas cortas.

**¿Qué sucede cuando el hijo recibe la señal de alarma?**
Al terminar la alarma manda la señal `SIGALRM`, pero al estar bloqueada durante la iteración

del bucle `for`, se espera a que termine y se desbloquee dicha señal. Esto lo hemos comprobado cambiando 40 por 10, ya que al ser 40 múltiplo de 8 podría haber ambigüedad entre si ha entrado justo al final del `sleep(3)` o al principio de la siguiente iteración.

## 5.2 Decisiones de diseño.

Hemos implementado la función `sigaddset_var(sigset_t *, int sig, ..., -1);` en la que agregamos al set tantas señales como queramos con la condición de que el último argumento sea -1. La definición de la función se encuentra en el archivo `mysignal.c`. Esto sirve para no tener que repetir la llamada a `sigaddset(2)` para cada señal que agreguemos al set.

## 5.3 Salida del programa.

Ejercicio 6a

```
$ ./build/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
```

# 6 Ejercicio 6b.

En este ejercicio se nos pide modificar un programa para que siga la especificación dada en la documentación de la práctica y contestar algunas cuestiones acerca de su ejecución.

## 6.1 Respuestas a preguntas cortas.

Al no haber bloqueado esta señal (`SIGTERM`) en ningún momento, el programa imprime lo requerido y termina su ejecución justo cuando recibe esta señal.

## 6.2 Salida del programa.

Ejercicio 6b

```
$ ./build/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
Soy 25723 y he recibido la senal SIGTERM
```

# 7 Ejercicio 8.

En este ejercicio se nos pide implementar la interfaz dada (`semaforos.h`) en C para crear nuestra librería de semáforos.

## 7.1 Decisiones de diseño.

Debido a que la interfaz no seguía la guía de estilo proporcionada, hemos modificado los nombres de las funciones para adaptarlos a ella. Además, hemos adaptado los comentarios de las funciones al estilo necesario para la ejecución de Doxygen.

# 8 Ejercicio 9.

En este ejercicio se nos pide que comuniquemos procesos padre-hijo por medio de ficheros, semáforos y señales con el objetivo de cumplir lo especificado en la documentación de la práctica.

## 8.1 Decisiones de diseño.

### 8.1.1 Ficheros.

Para cumplir con lo especificado en la práctica, el proceso padre genera una serie de ficheros con transacciones (que decidimos que fueran `floats` hasta 2 decimales entre 0 y 300 para ajustarse a la realidad).
Además, había que generar otra serie de archivos donde se guardaría el dinero de cada caja. En nuestro caso el responsable de crear este fichero es el propio hijo (ya que no hay mucha diferencia con la creación por parte del padre) y en vez de crear un fichero de texto creamos un fichero binario que almacena un `float` y un `bool`. Esto nos permite un uso más directo de las funciones `fread(3)` y `fwrite(3)`.

### 8.1.2 Semáforos.

En la práctica se nos pide el uso de la librería escrita en el ejercicio anterior. En nuestro caso hemos decidido utilizar un array de semáforos mutex de tamaño `NUM_CAJ`, de forma que cada uno controla el acceso a un fichero. De esta forma, el padre actuará sobre todos los semáforos y el hijo sólo sobre el que tenga su id. La key asignada al semáforo la generamos con la llamada a `ftok(``\bin \bash'', 2018)` con lo que nos aseguramos que el fichero existe.

### 8.1.3 Señales.

El proceso hijo ha de enviar una señal al padre para que recoja su dinero cuando éste esté por encima de 1000 euros. Esto lo hacemos mediante el envío de `SIGMONEY` (que es una macro de `SIGRTMIN`). De la misma forma, tienen que enviar otra señal cuando terminen, en este caso se envía `SIGDONE` (macro de `SIGRTMIN+1`). El uso de señales *realtime* se debe a

que estas funciones, a diferencia de las estándar de POSIX, se encolan cuando llegan más de una en vez de tratar la primera que llegue ignorando al resto (más información en la página 7 del manual de signal).

Estas señales son enviadas por medio de `sigqueue(3)` y recibidas por `sigaction(2)`. De esta forma, podemos enviar el id de la caja desde el proceso hijo al padre, siempre que pasemos la flag `SA_SIGINFO` en la llamada a `sigqueue(3)`. La recepción del entero se lleva a cabo mediante la correcta inicialización de la estructura necesaria en `sigaction(3)`.

Por último, usamos una máscara de señales tanto en la estructura de `sigaction(3)` como en la espera `sigsuspend(1)`. `sigaction(3)` bloqueará las señales de su máscara así como la señal que activó el manejador y la llamada a `sigsuspend(1)` captura las señales del set pasado como argumento.

Definimos 2 manejadores que se limitan a hacer lo pedido en la práctica.

### 8.1.4 Estructura del programa.

Para comenzar, aprovechando que no se dice lo contrario, hemos decidido usar memoria estática siempre que nos fuera posible ya que simplifica la programación y no es el objetivo principal de la práctica.

Hemos dividido el programa en distintas funciones, ya sea por la reutilización de código o por la legibilidad del mismo. Hemos creado la función `cajero(int id)`, que será la rutina que ejecute el proceso hijo una vez creado (de esta forma separamos el código del padre).

La creación de dos manejadores nos permite separar de forma lógica el código y las operaciones que realizamos dentro de estos nos *obliga* a definir ciertas variables globales. Tenemos 3 en total: el semid del semáforo (ya que tenemos que hacer down y up desde el manejador), el dinero total y el numero de cajeros terminados (por la modificación de los mismos desde el manejador). Además están declarados con la palabra clave `volatile`, lo que hará que el compilador no las optimice, cosa que nos interesa ya que se pueden modificar de forma asíncrona como resultado de capturar la señal.

Finalmente, las funciones `dinero_recogido(3)` y `increase_subtotal(2)` están separadas del resto del código por la reutilización del mismo.

### 8.1.5 Miscelánea/Dificultades.

En esta sección vamos a comentar algunas decisiones de diseño que no se pueden clasificar directamente en alguno de los apartados anteriores.

El uso del fichero de caja para almacenar un booleano es un *workaround* para evitar el uso de memoria compartida (que queda fuera del objetivo de la práctica). Gracias a este

booleano nos evitamos mandar señales cada vez que tengamos más de 1000 euros en la caja, y únicamente enviamos una.

El uso de las señales *realtime* se debe a que el programa no funcionaba siempre de la forma esperada con las señales POSIX `SIGUSR1` y `SIGUSR2`, ya que perdíamos la recepción de estas dos últimas señales en algunos casos (lo que impedía que terminase el programa). Esto es debido a la forma que tenemos de comunicar los procesos hijo con el padre. Creímos que esta era la forma óptima (ya que no tenemos que tener un fichero adicional para almacenar una cola) aunque para implementarla hayamos tenido que usar funciones como `sigaction(3)` y `sigqueue(3)`, no introducidas en la documentación de la práctica.

El día anterior a esta entrega el *Dr. Anguiano Rey* nos comentó que la práctica estaba orientada a la comunicación por medio de ficheros. Sin embargo, debido al poco tiempo existente decidimos mantener nuestra implementación (que no requiere el uso de una cola en un fichero, ya que está implícitamente implementada con el uso de señales *realtime*).

## 8.2 Salida del programa.

Ejercicio 6b

```
$ ./build/ejercicio6b

El total recaudado es: 23808.90 euros
```