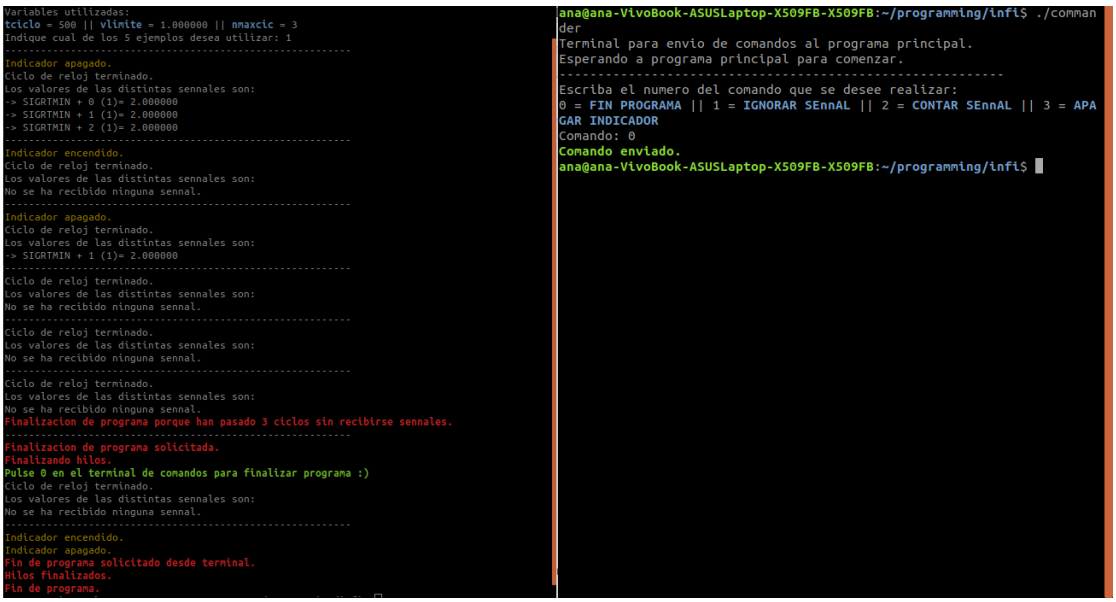
Informática Industrial

Trabajo para la Segunda Convocatoria

4º Grado en Ingeniería Electrónica, Robótica y Mecatrónica



Ana Mª Casado Faulí

Índice

1. Enunciado del Problema escogido (Examen del 10/6/13)

Resumen: Se pide realizar en C y con llamadas POSIX un programa multihilo que recibe los valores **N_SIG** señales físicas externas codificados en frecuencia, los decodifica y genera una alarma cuando se detecta una determinada condición. Las constantes simbólicas y declaraciones necesarias están definidas en la cabecera **problemas.h**, que se supone disponible.

Especificaciones detalladas:

- **Argumentos de la línea de comandos:** Los argumentos de la línea de comandos se interpretan como **tciclo** (argumento 1, entero que se expresa un número en milisegundos), **vlimite** (argumento 2, número fraccionario en la unidad de medida de las señales físicas, equivalente a número de señales por segundo), **nmaxcic** (argumento 3, entero que indica número de ciclos). Todos ellos están codificados en caracteres.
- **Recepción y decodificación de señales:** Las señales físicas externas se reciben mediante otras tantas señales POSIX desde **SIGRTMIN** a **SIGRTMIN+N_SIG-1**. La frecuencia de llegada de la señal **SIGRTMIN+i** expresada en número de señales por segundo es el valor de la señal física **i**-ésima. Para calcular tal valor se contarán las señales recibidas de cada tipo durante un periodo fijo, **tciclo**. Al final de cada ciclo de recepción se calculará cada uno de los **N_SIG** valores, expresados en **número de señales por segundo** y con tipo **float**.
- **Alarma:** La condición de alarma consiste en que el valor calculado en al menos ka mitad de las señales alcance o supere el valor **vlimite**. Cuando esto sucede se activa un indicador luminoso que permanece en ese estado (independientemente de los valores calculados en los ciclos posteriores) hasta que el proceso recibe una señal SIGRTMAX, que desactiva el indicador. La activación y la desactivación del indicador debe ser independiente de la recepción y decodificación de señales. Lógicamente si después de desactivar el indicador se detecta inmediatamente la condición de alarma, se tendrá que activar de nuevo. Para activar y desactivar el indicador se dispone de la función de biblioteca **void indicador(int valor)**.
- **Condiciones de fin:** Con cualquiera de las siguiente condiciones el proceso acaba, pero antes debe terminar la última decodificación de valores (si está en curso) y desactivar el indicador de alarma:
  - Llegada de una señal **SIGTERM**.
  - No llega ninguna señal durante al menos **nmaxcic**.

Otras condiciones:

- El ciclo debe realizarse con un **temporizador POSIX 1003.1b**
- El programa consistirá en **al menos dos hilos**, uno para gestionar el cálculo de los valores y otro para gestionar la activación y la desactivación de la alarma.
- El programa deberá funcionar **independientemente** de los valores concretos de los argumentos y las constantes simbólicas.
- Es necesario utilizar mutex y variables de condición según lo indicado en clase para gestionar el acceso y la sincronización de datos compartidos.
- Se recomienda tratar con señales síncronamente, aunque se admiten otras soluciones.
- No es necesario considerar tratamiento de errores.
- Es necesario acompañar el programa de pseudocódigo o una explicación de su funcionamiento.

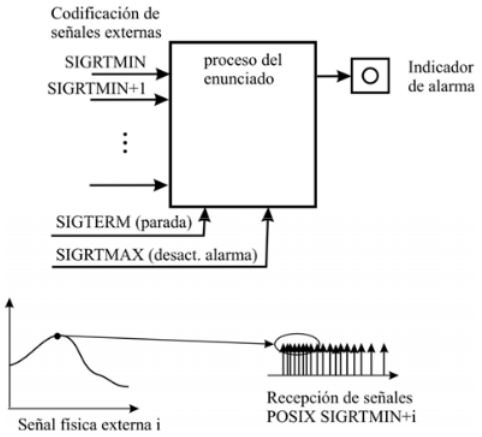
Fichero de cabecera:

```
/* Cabecera problema.h */

#define N_SIG 4

/* Función para activar y desactivar el indicador de alarma
   (valor=0: desactivar, valor=1: activar */

void indicador(int valor);
```



2. Consideraciones Respecto al Enunciado

Al no quedar del todo claro, se ha decidido lo siguiente respecto a algunas cuestiones:

- Se debe superar **nmaxcic** en varios ciclos seguidos, es decir, si en un ciclo no llega ninguna señal pero en el siguiente sí se vuelve a empezar la cuenta de ciclos sin señales.
- Para activar el indicador, la suma de señales que superan la mitad de **N_SIG** se hace en cada ciclo. No se acumula de ciclo a ciclo.
- Para la función indicador se ha creado un archivo problema.c donde esta función consiste en imprimir el estado de este (encendido o apagado).

3. Otras funcionalidades añadidas para mejorar el programa

Se ha decidido crear una funcionalidad de envío de comando a través de terminal, para la cual se ha creado una estructura de servidor/cliente. En la siguiente sección de explica que comandos son los que se han diseñado.

4. Estructura del sistema

Para el funcionamiento de este sistema se ha diseñado un servidor y un cliente que mantienen una comunicación entre ellos. Se ha decidido hacer así ya que se usará un terminal para el envío de comandos y otro para la impresión por pantalla de información sobre el funcionamiento del sistema descrito en el enunciado del problema. Para su diseño se ha usado el sistema operativo Ubuntu 18 pero se puede probar su funcionamiento en cualquier otra versión ya que se usan funciones de POSIX y librerias que tienen otras versiones de Ubuntu.

En la siguiente sección se presenta el código de todos los archivos con comentarios que explican el funcionamiento de cada parte del programa. Pero primero, un resumen de cada archivo:

- *solucion.c*: en este archivo se encuentra todo el sistema definido por el problema. Es donde se realiza la recepción, cuenta y cálculo de las señales que se reciben de tipo **SIGRTMIN + i**, la gestión de la alarma y donde se gestiona la recepción de mensajes de comando que se envían desde terminal. El programa se compone de las siguientes funciones:

  - `main`: este es el programa principal. Primero se recogen y guardan las variables que pide el programa, luego enmascara las señales que van a recibir los hilos y crea los hilos. Por último, espera la finalización del programa que llega con la señal **SIGTERM**, tras lo cual, finaliza los hilos enviando las señales necesarias para acabar con las esperas de estos, excepto el hilo de comandos, que debe terminarse desde el terminal de comandos para sincronizar el final de programa de ambos terminales.
  - `sigCalc`: esta es la función asociada al hilo de gestión de las señales físicas. Primero, se define el temporizador con la variable **tciclo** y pidiendo que se envíe **SIGALRM** cada ciclo de reloj del temporizador. Se define que se reciban **SIGRTMIN + i** y **SIGALRM** y se espera a que llegue una de estas. Si llegan señales del primer tipo, se suma que ha llegado en ese ciclo de reloj (teniendo en cuenta si esa señal se ha decidido bloquear o no). Si llega la señal **SIGALRM** se calcula cuántas señales del primer tipo han llegado para calcular el valor físico como define el problema. Si no llega ninguna señal del primer tipo, se cuenta que en ese ciclo no han llegado y si se sobrepasan **nmaxcic** ciclos sin señales se envía **SIGTERM** para finalizar el programa. Además, si el valor físico de la mitad de las señales sobrepasa **vlimite** se activará la variable de condición para encender la alarma.
  - `alarmManage`: Esta es la función asociada al hilo de gestión de la alarma. El indicador se encenderá si se recibe el cumplimiento de la variable de condición definida en el hilo anterior. Se apagará el indicador si se recibe la señal **SIGRTMAX**.
  - `messageReceiver`: Esta es la función asociada al hilo de recepción de mensajes. Se utilizan colas de mensajes para pasar información de un programa a otro. Está diseñado como un cliente (que es una cola de mensajes), por lo que primero tiene que establecer la conexión con el servidor (que es otra cola de mensajes) identificandose. Esto se hace enviandose y recibiendo mensajes para comprobar la comunicación. Una vez hecho esto se mantiene la conexión, en la que el servidor (el programa de *commander.c*) enviará los comandos. Estos comandos son números que dependiendo su valor ejecutan una acción u otra. Dichas acciones son: terminar el programa, enviando la señal **SIGTERM**, ignorar en el conteo una de las señales, volver a añadir una señal bloqueada y apagar el indicador, enviando la señal **SIGRTMAX**.
  - `signalExamples`: esta es la función asociada al hilo de ejecución de los ejemplos. Estos son distintos patrones de señales diseñados para probar distintas partes del programa. Vienen todos explicados en el mismo código. El ejemplo se elige al principio de la ejecución por terminal.

- *commander.c*: este programa está diseñado como un servidor. Utiliza colas de mensajes para comunicarse con el cliente, como se explicó antes. Si el comando es ignorar o añadir una señal, se deberá enviar además el valor de dicha señal (se escribe desde terminal dicho valor cuando se pide).

- *problema.c* y *problema.h*: el .h es la cabecera dada por el problema y el .c no es más que un `if` que imprime si el indicador se enciende o se apaga.

- *signalMessagesExample.c y signalMessagesExample.h*: donde se encuentran los patrones de los envíos de señales.

5. Cómo ejecutar el programa

Si se desea recompilar los ejecutables se deben usar los siguientes comandos:

```
$ gcc solucion.c problema.c signalMessagesExample.c -o solucion -lpthread -lrt
$ gcc commander.c -o commander -lrt
```

Para ejecutarlos es necesario que se usen 2 terminales diferentes, uno para `./solucion` y otro para `./commander`. Además, para el ejecutable `./solucion` es necesario escribir los valores de **tciclo**, **vlimite** y **nmaxcic** en ese orden. Un ejemplo de ejecución podría ser:

```
$ ./solucion 500 1 3
```

6. Algunos ejemplos de funcionamiento

A continuación, se muestran los ejemplos más interesantes de los diseñados para hacer pruebas, que son el 1 y el 4. Cuando se ejecutan `./solucion` y `./commander`, se debe tener algo así:

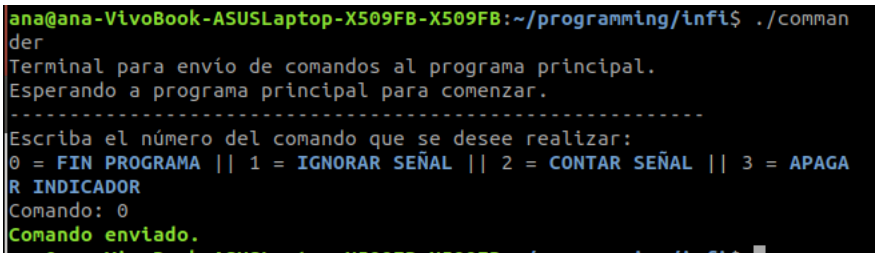
El terminal del programa principal espera a que indiquemos que ejemplo queremos utilizar y el de comandos espera a que el programa principal inicialice el cliente (que se hace luego al iniciar los hilos).

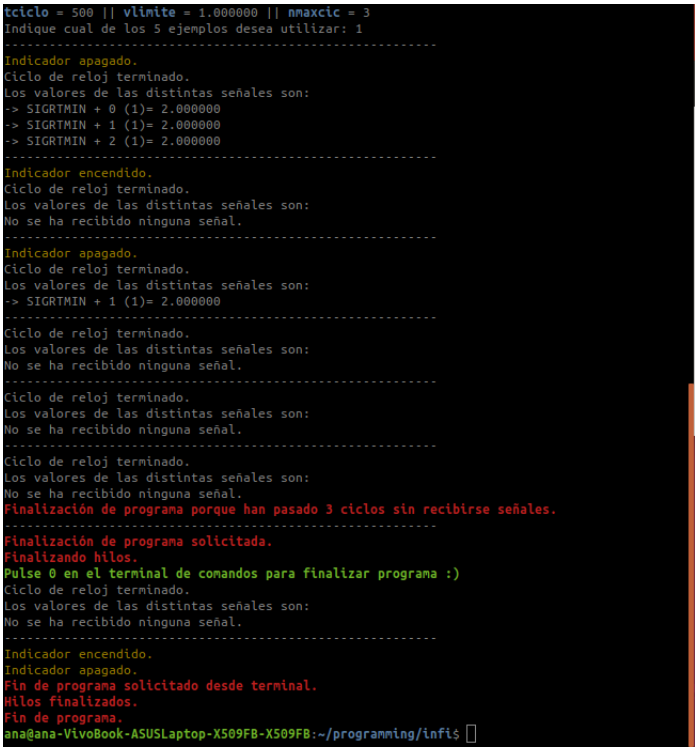
6.1. Ejemplo 1

El primer ejemplo es el envío de las siguientes señales:

```
6   void testExample1(){
7       kill(getpid(), SIGRTMIN+1);
8       kill(getpid(), SIGRTMIN);
9       kill(getpid(), SIGRTMIN+2);
0       sleep(1);
1       kill(getpid(), SIGRTMIN+1);
2       kill(getpid(), SIGRTMAX);
3   }
```

Si no se toca el terminal de comando (antes de que el programa lo pida al final, que se debe pulsar 0 para acabar) se ejecuta lo siguiente:





Como se sobrepasa el valor de **vlimite** se enciende el indicador y se apaga luego ya que se recibe la señal **SIGRTMAX**. Pasan 3 ciclos de reloj sin señales y el programa empieza a cerrar los hilos. Finalmente, espera a que se pulse 0 desde el terminal de comandos para finalizar completamente.

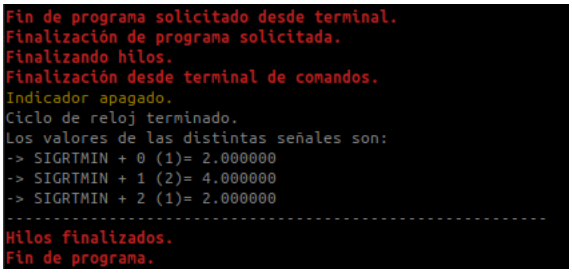
6.2. Ejemplo 4

Es un envío cíclico de señales para hacer pruebas con el terminal de comandos.

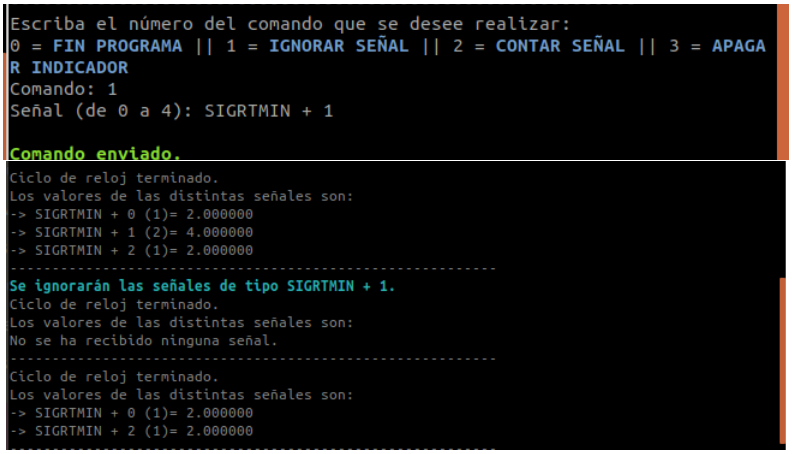
```c
case 4:
{
    // Envía un patrón de señales cada segundo.
    // Diseñado para probar los comandos y que se vea mejor la activación
    // y desactivación de la alarma (porque el segundo ejemplo es demasiado rápido).
    while (end!=1)
    {
        testExample4();
    }
    break;
}
```

```c
void testExample4(){
    kill(getpid(), SIGRTMIN+1);
    kill(getpid(), SIGRTMIN);
    kill(getpid(), SIGRTMIN+2);
    sleep(1);
    kill(getpid(), SIGRTMIN+1);
}
```

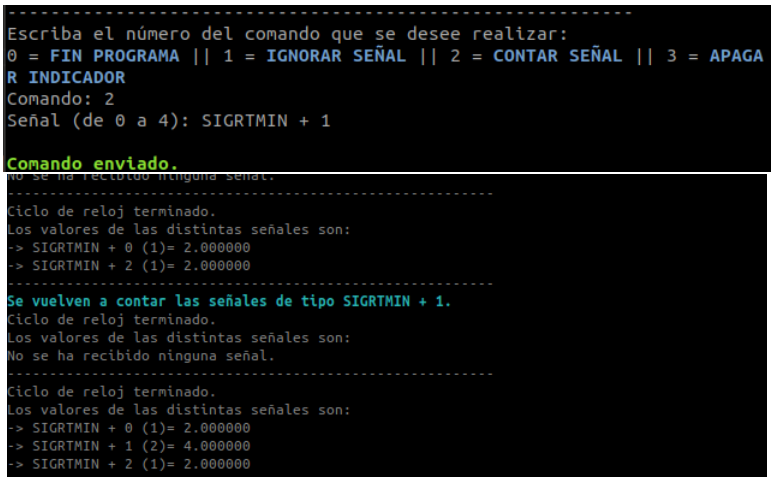
- Al pulsar 0:



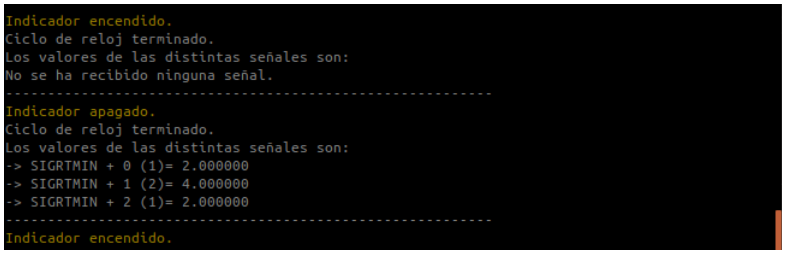
- Al pulsar 1 (si la señal no estaba ya bloqueada):



- Al pulsar 2 (si la señal no estaba ya añadida):



- Al pulsar 3:



7. Código comentado

A continuación se tiene todo el código realizado para *solcuion.c* y *commander.c*, que es donde se encuentra lo principal de las aplicaciones diseñadas.

7.1. *solucion.c*

```
#define _POSIX_C_SOURCE 199506L     // POSIX 1003.1c para hilos.

#include <unistd.h>                 // POSIX.
#include <stdlib.h>                 // Macros.
#include <stdio.h>                  // Input/output para sscanf y printf.
#include <time.h>                   // Temporizadores.
#include <string.h>                 // Para tratar los string.
#include <pthread.h>                // Hilos.
#include <signal.h>                 // Sennales.
#include <mqueue.h>                 // Para las colas de mensajes
#include <sys/stat.h>               // Macros.

#define COMMANDER   "/commander-queue"      // Nombre de la cola de comandos (la declaramos aqui para que
                                            // tenga el mismo nombre en el otro programa)
#define MAX_MESSAGES 10                     // Maximo num. de mensajes para la cola
#define MAX_MSG_SIZE 256                    // Tamanno maximo de mensaje
#define MSG_BUFFER_SIZE MAX_MSG_SIZE + 10   // Tamanno de mensaje en buffer
                                            // (le sumamos 10 al maximo para darle margen)

#include "problema.h"               // Cabecera dada por problema.
#include "signalMessagesExample.h"  // Cabecera de ejemplos de envios de sennales

// Variables por linea de comando (necesario que sean globales porque se usan en los hilos
// pero no cambiaran de valor durante la ejecucion por lo que no haran falta mutex para su acceso).
int tciclo;     // Num. de milisegundos por ciclo.
float vlimite;  // Maximo valor que pueden tener las sennales fisicas (num. sennales/segundo).
int nmaxcic;    // Num. de ciclos maximo que podemos estar sin recibir sennales de tipo SIGRTMIN+i.
int example;    // Ejemplo que se desea utilizar

// Otras variables compartidas que si requeriran mutex porque
// variaran su valor durante la ejecucion del programa
int overflowedSignals = 0;  // Num. de sennales que tienen valor mayor que vlimite.
int blockSignal[N_SIG];     // Matriz para la gestion de sennales que se deben ignorar

// Temporizador POSIX 1003.1b (declarado absoluto para recoger el tiempo absoluto).
timer_t timerDef;

// Flag de fin de programa (para que los hilos lleguen a su fin y se pueda hacer join).
int end = 0;
// Flag para indicar si se ha pedido fin de programa por linea de comando o no
int out = 0;

// Mutex y variables de condicion asociadas a la variable overflowedSignals.
pthread_mutex_t mut_overflowedSignals       = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv_overflowedSignals        = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut_blockSignal             = PTHREAD_MUTEX_INITIALIZER;

// Manejador en el que nunca entra pero necesario para la definicion.
void handeling(int signo, siginfo_t *info, void *p){}

// Definicion de las funciones de los hilos.
void *signalCalc(void *p);
void *alarmManage(void *p);
void *messageReceiver(void *p);
void *signalExamples(void *p);

////////////////////////////////////////// MAIN ////////////////////////////////////////////
// Se encarga de recoger las variables del problema, gestionar los hilos y sus sennales y
// de recibir el fin del programa (con la recepcion de la sennal SIGTERM).
int main(int argc, char **_argv){

    // Guardar argumentos de la linea de comandos.
    printf("Variables utilizadas:\n");
    sscanf(_argv[1], "%d", &tciclo);    printf("\033[1;34mtciclo\033[0m = %d || ", tciclo);
    sscanf(_argv[2], "%f", &vlimite);   printf("\033[1;34mvlimite\033[0m = %f || ", vlimite);
    sscanf(_argv[3], "%d", &nmaxcic);   printf("\033[1;34mnmaxcic\033[0m = %d\n", nmaxcic);

    // Escoger ejemplo por linea de comandos.
    printf("Indique cual de los %d ejemplos desea utilizar: ", N_EXAMPLES);
```

```
    scanf("%d", &example);
    printf("------------------------------------------------------------\n");

    // Recepcion de las sennales (el manejador no se usa ya que usaremos sigwaitinfo).
    struct sigaction act;
    act.sa_flags=SA_SIGINFO;    // Usando 1003.1b hay que usar este flag para asegurar funcionamiento.
    act.sa_sigaction = handeling;  // Manejador.
    sigemptyset(&act.sa_mask);  // No se usa mascara.

    // Definicion del conjunto de sennales que se espera recibir.
    sigset_t expectedSignals;
    // Inicializacion del conjuto.
    sigemptyset(&expectedSignals);

    for(int i=0; i<N_SIG; i++){
        sigaction(SIGRTMIN+i, &act, NULL);          // Asociamos la accion (recibirlas) a las sennales.
        sigaddset(&expectedSignals, SIGRTMIN+i);    // Annadimos al conjunto las sennales SIGRTMIN+i.
        blockSignal[i] = 0;                         // Inicializamos con 0 la matriz de bloqueo de sennales.
    }
    // Annadimos SIGRTMAX y SIGALRM al conjunto.
    sigaddset(&expectedSignals, SIGRTMAX);
    sigaddset(&expectedSignals, SIGALRM);

    // Mascara utilizada para bloquear las sennales hasta ahora annadidas a expectedSignals
    // para que solo las puedan usar en sigwaitinfo en los hilos correspondientes
    // (si no se hiciese esto el programa no funciona correctamente).
    pthread_sigmask(SIG_BLOCK, &expectedSignals, NULL);

    // Variables que reciben el identificador de los hilos.
    pthread_t sC;   // signalCalc.
    pthread_t aM;   // alarmManage.
    pthread_t mR;   // messageReceiver
    pthread_t sE;   // signalExamples

    // Creamos los hilos.
    pthread_create(&sC, NULL, signalCalc, NULL);
    pthread_create(&aM, NULL, alarmManage, NULL);
    pthread_create(&mR, NULL, messageReceiver, NULL);
    pthread_create(&sE, NULL, signalExamples,NULL);

    // Reutilizamos el conjunto para procesar otras (ya que las anteriores han sido ya bloqueadas).
    sigemptyset(&expectedSignals);
    // Asociamos la accion de recepcion a la sennal SIGTERM.
    sigaction(SIGTERM, &act, NULL);
    // Metemos SIGTERM en el conjunto.
    sigaddset(&expectedSignals, SIGTERM);
    // Esta sennal no la bloqueamos porque no la recibiran los hilos, sino el programa principal.

    // El programa principal espera hasta que recibe una sennal de tipo SIGTERM (que se puede recibir
    // como sennal independiente o del hilo signalCalc si no se reciben sennales en un tiempo determinado).
    siginfo_t info; // Necesario para definir sigwaitinfo.
    sigwaitinfo(&expectedSignals, &info);
    printf("\033[1;31mFinalizacion de programa solicitada.\n\033[0m");

    // Finalizamos los hilos haciendo que salgan de sus respectivos bucles while activando el flag end.
    end=1;
    printf("\033[1;31mFinalizando hilos.\n\033[0m");

    // Para finalizar el hilo asociado a signalCalc forzamos la sennal de fin de ciclo del
    // temporizador (SIGALRM) (aunque si se esta en medio de una decodificacion primero terminara eso).
    kill(getpid(), SIGALRM);

    // Para finalizar el hilo asociado a alarmManage enviamos SIGRTMAX, para terminar la espera en
    // sigwaitinfo, y por tanto se apaga el indicador (si estaba apagada se encendera y luego se apagara).
    kill(getpid(), SIGRTMAX);

    // El programa se queda sin finalizar el hilo de messageReceiver si llega SIGTERM desde una orden
    // fuera del terminal de comandos. Por ello es necesario que desde dicho terminal se envie el mensaje de
    // fin de programa (0) y para ello se pide con este print (se vera en verde). Esta solucion
    // sincroniza ademas que se cierre el terminal de comandos junto al programa principal.
    if(out == 0){
        printf("\033[1;32mPulse 0 en el terminal de comandos para finalizar programa :)\n\033[0m");
    }
    else{
        printf("\033[1;31mFinalizacion desde terminal de comandos.\n\033[0m");
    }

    // Espera de la finalizacion de los hilos.
    pthread_join(sC, NULL);
```

```
    pthread_join(aM, NULL);
    pthread_join(mR, NULL);
    pthread_join(sE, NULL);

    printf("\033[1;31mHilos finalizados.\nFin de programa.\n\033[0m");

    return 0;
}

//////////////////////////// HILO CALCULO VALORES DE LAS SENNALES ////////////////////////////
// Este es el hilo principal del programa. Es el encargado de calcular los valores de las
// sennales fisicas externas que vienen representadas por la frecuencia de recepcion de sennales
// de tipo SIRTMIN+i. Para ello se tiene que definir un temporizador que nos marcara ciclos
// de tiempo que usaremos para el calculo de la frecuencia con la que llegan las sennales.
void *signalCalc(void *p){
    int receivedSignal;         // Valor de la sennal que se recibira a traves de sigwaitinfo.
    int signalCount[N_SIG];     // Vector que cuenta cuantas sennales de cada tipo han llegado.
    float signalValue[N_SIG];   // Vector con el calculo del valor fisico de cada una de las sennales.
    int noSignal = 0;           // Contador de cuantas sennales no se ha recibido en un ciclo de reloj.
    int emptyCycle = 0;         // Num. de ciclos seguidos en los que no se ha recibido ninguna sennal.

    // Inicializamos valores a 0
    for(int i=0; i<N_SIG; i++){
        signalCount[i]=0;
        signalValue[i]=0.0;
    }

    // Variable necesaria para usar sigwaitinfo.
    siginfo_t in;

    // Si los milisegundos que se nos dan equivalen a mas de 1000 ms es necesario dividir el tiempo
    // para el temporizador (porque tiene variable en segundos y variable en nanosegundos).
    int seconds = 0;
    int millisec = tciclo;
    if (millisec>=1000){
        seconds = (int)(millisec/1000);
        millisec = millisec%1000;
    } // seconds*1000 + millisec = tciclo.

    // Especificaciones de tiempo del temporizador.
    struct timespec clock;
    clock.tv_sec = seconds;             // Segundos.
    clock.tv_nsec = millisec*1000000;   // Nanosegundos.

    // Comportamiento temporizador.
    struct itimerspec cycle;
    cycle.it_value=clock;       // Primer disparo a los tciclo milisegundos.
    cycle.it_interval=clock;    // Ciclo de tciclo milisegundos.

    // El vencimiento del temporizador (cuando ha pasado un ciclo) avisara con una sennal SIGALRM.
    struct sigevent event;
    event.sigev_signo = SIGALRM;
    event.sigev_notify = SIGEV_SIGNAL;

    // Creacion del temporizador con el evento programado.
    timer_create(CLOCK_REALTIME, &event, &timerDef);
    // Programacion del temporizador con el comportamiento programado.
    timer_settime(timerDef, 0, &cycle, NULL);

    // Inicializacion del conjunto de sennales signalSet.
    sigset_t signalSet;
    sigemptyset(&signalSet); // Para inicializar.
    // Annadir las sennales que se van a recibir de tipo SIGRTMIN+i.
    for(int i=0; i<N_SIG; i++){
        sigaddset(&signalSet, SIGRTMIN+i);
    }
    // Annadimos SIGALRM al conjunto de sennales que podemos recibir
    // (que representa el fin de un ciclo del temporizador).
    sigaddset(&signalSet, SIGALRM);

    while(end!=1){ // Mientras no sea el final del programa.
        // Guardamos el valor de la sennal que recibe sigwaitinfo.
        receivedSignal=sigwaitinfo(&signalSet, &in);

        if(receivedSignal==SIGALRM){ // Si la sennal que se recibe es un fin de ciclo de reloj.
            printf("Ciclo de reloj terminado.\n");

            noSignal = 0; // Inicializamos contador de sennales no recibidas.
```

```
        // overflowedSignals es una variable compartida con el hilo asociado a alarmManage por
        // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
        pthread_mutex_lock(&mut_overflowedSignals);
        overflowedSignals = 0;  // Inicializamos contador de sennales que sobrepasan el valor vlimite.
        // Liberamos mutex ya que hemos terminado de usar la variable.
        pthread_mutex_unlock(&mut_overflowedSignals);

        // Se calcula el valor de las sennales dependiendo de la frecuencia con la que hayan llegado.
        printf("Los valores de las distintas sennales son:\n");
        for(int i=0; i<N_SIG; i++){ // Vamos a ver cada una de las sennales SIGRTMIN+i por separado.

            if(signalCount[i]==0){ // Si no se han recibido sennales de este tipo.
                // Contamos que uno de los tipos de sennales no ha llegado.
                noSignal++;

                if(noSignal==N_SIG){ // Si no ha llegado ninguna sennal en este ciclo de reloj.
                    printf("No se ha recibido ninguna sennal.\n");
                    emptyCycle++;

                    if(emptyCycle==nmaxcic){ // Si el num. de ciclos vacios es mayor al maximo permitido
.
                        printf("\033[1;31mFinalizacion de programa porque han pasado %i ciclos sin
recibirse sennales.\n\033[0m", nmaxcic);
                        kill(getpid(),SIGTERM); // Se envia la sennal SIGTERM para terminar el programa.
                    }
                }
            }
            else{ // Se han recibido sennales del tipo que corresponde a i.

                noSignal = 0;   // Como se ha recibido una sennal, lo ponemos a 0.
                emptyCycle = 0; // y se vuelve a empezar la cuenta de ciclos vacios.

                // DECODIFICACIoN:
                // El valor de la sennal fisica es igual a la frecuencia con la que
                // se ha recibido la sennal del tipo SIGRTMIN+i en un ciclo de reloj
                // (es decir, cuantas sennales se han recibido por ciclo de reloj).
                // El casteo es necesario ya que la mayoria de las variables son int.
                // Se divide millisec/1000 porque lo queremos en segundos.
                signalValue[i] = (float)signalCount[i]/((float)seconds+(float)millisec/1000.0);
                printf("-> SIGRTMIN + %i (%d)= %f\n", i, signalCount[i], signalValue[i]);

                // overflowedSignals es una variable compartida con el hilo asociado a alarmManage por
                // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
                pthread_mutex_lock(&mut_overflowedSignals);
                if(signalValue[i]>=vlimite){ // Si la sennal iguala o sobrepasa el valor limite.
                    overflowedSignals++; // Contamos esa sennal como que ha sobrepasado.

                    // Si el num. de sennales que sobrepasan vlimite es mayor que la mitad
                    // del num. de sennales totales entraremos en este if.
                    // Para evitar problemas con las divisiones, en vez de escribir la condicion
                    // literalmente como la frase (overflowedSignals >= N_SIG/2)
                    // pasamos el 2 dividiendo al otro lado (multiplicando) y tenemos lo mismo.
                    if(overflowedSignals*2 >=N_SIG){
                        // Mandamos que la condicion se ha cumplido y asi el hilo asociado a alarmManage
                        // puede continuar (en este caso para encender el indicador).
                        pthread_cond_signal(&cv_overflowedSignals);
                    }
                }
                // Liberamos el mutex porque ya hemos terminado de usar overflowedSignals.
                pthread_mutex_unlock(&mut_overflowedSignals);
            }
        }

        // Ya se han hecho los calculos en este ciclo de reloj.
        // Para empezar la cuenta de nuevo ponemos todo a 0
        for(int i=0; i<N_SIG; i++){
            signalCount[i]=0;
            signalValue[i]=0.0;
        }
        printf("------------------------------------------------------------\n");
    }
    else{ // Si no es SIGALRM sera una de las sennales de tipo SIGRTMIN + i.
        // blockSignals es una variable compartida con el hilo asociado a messageReceiver por
        // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
        pthread_mutex_lock(&mut_blockSignal);
        // Desde el hilo de messageReceiver se deciden que sennales se cuentan y que sennales no.
        // Si blockSignal[num. de la sennal] = 1, la sennal no se debe contar en este momento.
        if (blockSignal[receivedSignal-SIGRTMIN]!=1){
                // Cada posicion del vector esta asociada a una de las sennales por lo que si
```

```
                    // restamos SIGRTMIN podemos ordenar las sennales como 0, 1, ... Este vector
                    // nos sirve para saber cuantas veces han llegado las sennales de cada tipo antes de que
                    // acabe un ciclo de reloj (sumando 1 cada vez que llega una de las sennales).
                    signalCount[receivedSignal-SIGRTMIN]++;
            }
            // Liberamos el mutex porque ya se ha terminado de usar blockSignal.
            pthread_mutex_unlock(&mut_blockSignal);
        }
    }

    // overflowedSignals es una variable compartida con el hilo asociado a alarmManage por
    // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
    pthread_mutex_lock(&mut_overflowedSignals);
    // Para terminar el programa hay que asegurarse que alarma no se quede esperando
    // en la variable de condicion por lo que hacemos que overflowedSignals tenga
    // un valor exageradamente grande para que salga del while.
    overflowedSignals=1000;
    // Ademas enviamos que la variable de condicion se ha cumplido para que desbloqueee
    // de la espera.
    pthread_cond_signal(&cv_overflowedSignals);
    // Liberamos el mutex porque ya hemos terminado de usar overflowedSignal.
    pthread_mutex_unlock(&mut_overflowedSignals);
}

////////////////////////////////////// HILO ALARMA //////////////////////////////////////
// Hilo de gestion de la alarma. Dependiendo de la condicion establecida
// se encendera o apagara un indicador (programado en problema.c/.h).
void *alarmManage(void *p){
    //Inicializamos el indicador apagado.
    indicador(0);
    // Creamos el conjunto de sennales que contendra
    // la sennal SIGRTMAX que es la sennal que apaga el indicador.
    sigset_t turnOff;
    sigemptyset(&turnOff); // Para inicializar.
    sigaddset(&turnOff, SIGRTMAX);

    // Variable necesaria para sigwaitinfo.
    siginfo_t info;

    //Mientras no sea el fin del programa.
    while(end!= 1){
        // overflowedSignals es una variable compartida con el hilo asociado a signalCalc por
        // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
        pthread_mutex_lock(&mut_overflowedSignals);

        // Mientras el num. de sennales que sobrepasan vlimite no sea mayor que la mitad
        // del num. de sennales totales seguiremos en este bucle.
        // Para evitar problemas con las divisiones, en vez de escribir la condicion
        // literalmente como la frase (overflowedSignals < N_SIG/2)
        // pasamos el 2 dividiendo al otro lado (multiplicando) y tenemos lo mismo.
        while(overflowedSignals*2 < N_SIG){
            // Esta condicion tambien se verifica en el otro hilo por lo que esperamos a recibir
            // la variable de condicion para continuar. Se saldra del while y se continuara a
            // encender el indicador.
            pthread_cond_wait(&cv_overflowedSignals, &mut_overflowedSignals);
        }

        // Como se ha cumplido la condicion encendemos el indicador.
        indicador(1);
        // Liberamos el mutex porque ya hemos terminado de usar overflowedSignals.
        pthread_mutex_unlock(&mut_overflowedSignals);

        // Esperamos que llegue la sennal SIGRTMAX.
        sigwaitinfo(&turnOff, &info);

        // Una vez llega la sennal podemos apagar el indicador.
        indicador(0);
    }
}

////////////////////////////////////// HILO RECIBE MENSAJES //////////////////////////////////////
// Hilo de tipo cliente. Se encargara de recibir mensajes de un servidor (terminal de comando)
// y dependiendo del contenido del mensaje se realizaran distintas acciones.
void *messageReceiver(void *p){
    mqd_t receiver, commander;  // Id de las colas de mensajes.
    char receiverName[64];      // Nombre de la cola que recibe los comandos.

    // Guardamos el nombre de la cola que recibe datos. El nombre depende del valor del
    // id del proceso porque se conecta a un servidor que puede recibir mas clientes.
```

```
sprintf(receiverName, "/receiver-queue-%d", getpid());

 //Definimos la caracteristicas de la cola.
struct mq_attr attr;
attr.mq_flags   = 0;
attr.mq_maxmsg  = MAX_MESSAGES;
attr.mq_msgsize = MAX_MSG_SIZE;
attr.mq_curmsgs = 0;

// Abrimos la cola que usaremos aqui que es la que recibe los comandos desde el terminal.
// Se creara la cola si no existe y sino se leera. Ademas damos acceso total de lectura, escritura
// y ejecucion, y por ultimo le asociamos los atributos anteriores.
// El uso del if es para deteccion de fallos de comunicacion (para debug).
if ((receiver = mq_open(receiverName, O_RDONLY | O_CREAT, S_IRWXU, &attr)) == -1) {
    printf("\033[1;31mFin de programa por error al crear recepcion de mensajes.\n\033[0m");
    kill(getpid(), SIGTERM);
}

// Abrimos la cola de la que recibiremos los comandos (por eso solo tenemos permisos de escritura).
// El uso del if es para deteccion de fallos de comunicacion (para debug).
if ((commander= mq_open(COMMANDER, O_WRONLY)) == -1){
    printf("\033[1;31mFin de programa por no poder contactar programa de comandos.\n\033[0m");
    kill(getpid(), SIGTERM);
}

char inputBuffer[MSG_BUFFER_SIZE];  // Buffer de datos de entrada.
int commandReceived;                // Variable que contendra el comando a ejecutar.

// Mientras no sea el fin del programa.
while(end!=1){
    // Enviamos el nombre de la cola que va a recibir los comando a la que los va
    // a enviar. Se hace asi porque el disenno del terminal de comandos es muy parecido
    // al de un servidor que puede recibir distintos clientes y es necesario que
    // cada cliente se ejecute separadamente en cada intercambio de mensajes.
    // El tamanno del "buffer" en este caso no es mas que la longitud del nombre
    // de la cola porque no es necesario mas.
    // El uso del if es para deteccion de fallos de comunicacion (para debug).
    if (mq_send (commander, receiverName, strlen(receiverName)+1, 0) == -1) {
        printf("\033[1;31mFin de programa por no contactar con comandos.\n033[0m");
        kill(getpid(), SIGTERM);
    }

    // Aqui recibimos el comando del terminal dentro de inputBuffer.
    // El uso del if es para deteccion de fallos de comunicacion (para debug).
    if (mq_receive (receiver, inputBuffer, MSG_BUFFER_SIZE, NULL) == -1) {
        printf("\033[1;31mFin de programa por mala recepcion.\n\033[0m");
        kill(getpid(), SIGTERM);
    }

    // Leemos el valor del comando, ya que el buffer se interpreta como caracteres.
    sscanf(inputBuffer, "%d", &commandReceived);

    // Dependiendo del valor se ejecutan comandos differentes.
    switch (commandReceived){
        case 0:
        {
            // FIN DE PROGRAMA:
            // Se envia la sennal de fin de programa SIGTERM que se espera en el main.
            printf("\033[1;31mFin de programa solicitado desde terminal.\n\033[0m");
            out = 1; // Flag para indicar que se termina desde aqui el programa.
            kill(getpid(), SIGTERM);
            sleep(1);   // Para asegurar que la sennal se envia,
                        //cambia el flag de end y no se quede el
                        // programa colgado.
            break;
        }
        case 1:
        {
            // IGNORAR SEnnAL:
            // Desde el terminal de comandos se recibe que sennal se desea no contar para el calculo.
            // Se recibe dicho valor en inputBuffer.
            // El uso del if es para deteccion de fallos de comunicacion (para debug).
            int signal;
            if (mq_receive (receiver, inputBuffer, MSG_BUFFER_SIZE, NULL) == -1) {
                printf("\033[1;31mFin de programa por mala recepcion.\n\033[0m");
                kill(getpid(), SIGTERM);
            }
            sscanf(inputBuffer, "%d", &signal); // Leemos la sennal que se quiere ignorar.
```

```
                // blockSignals es una variable compartida con el hilo asociado a signalCalc por
                // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
                pthread_mutex_lock(&mut_blockSignal);
                if (blockSignal[signal]!=1){ // Comprobamos que no este ya bloqueada.
                    blockSignal[signal]=1; // Indicamos que se debe bloquear.
                    printf("\033[1;36mSe ignoraran las sennales de tipo SIGRTMIN + %d.\n\033[0m", signal);
                }
                else {
                    printf("\033[1;36mSennal ya ignorada.\n\033[0m");
                }
                // Liberamos el mutex porque se ha dejado de usar blockSignal.
                pthread_mutex_unlock(&mut_blockSignal);

                break;
            }
            case 2:
            {
                // CONTAR SENNAL:
                // Desde el terminal de comandos se recibe que sennal se desea volver a contar
                // para el calculo. Se recibe dicho valor en inputBuffer.
                // El uso del if es para deteccion de fallos de comunicacion (para debug).
                int signal;
                if (mq_receive (receiver, inputBuffer, MSG_BUFFER_SIZE, NULL) == -1) {
                    printf("\033[1;31mFin de programa por mala recepcion.\n\033[0m");
                    kill(getpid(), SIGTERM);
                }
                sscanf(inputBuffer, "%d", &signal); //Leemos la sennal que se quiere contar

                // blockSignals es una variable compartida con el hilo asociado a signalCalc por
                // lo que es necesario usar mutex para bloquear el uso simultaneo de esta variable.
                pthread_mutex_lock(&mut_blockSignal);
                if(blockSignal[signal]!=0){ // Comporbamos que no se este contando.
                    blockSignal[signal]=0; // Indicamos que se debe contar
                    printf("\033[1;36mSe vuelven a contar las sennales de tipo SIGRTMIN + %d.\n\033[0m",
    signal);
                }
                else{
                    printf("\033[1;36mSennal ya contada.\n\033[0m");
                }
                // Liberamos el mutex porque se ha dejado de usar blockSignal.
                pthread_mutex_unlock(&mut_blockSignal);

                break;
            }
            case 3:
            {
                // APAGAR INDICADOR:
                // Se envia la sennal que apaga el indicador.
                kill(getpid(), SIGRTMAX);
                break;
            }
        }
    }

    // Cerramos la cola que recibe mensajes.
    if (mq_close (receiver) == -1) {
        perror ("Error al cerrar.\n"); // Para debug.
        exit (1);
    }
    // Desvinculamos la cola.
    if (mq_unlink (receiverName) == -1) {
        perror ("Error al cerrar.\n"); // Para debug.
        exit (1);
    }
}

////////////////////////////////////// HILO EJEMPLOS ////////////////////////////////////////
// Hilo que llama a varias funciones que contienen envio de sennales.
// El ejemplo a utilizar se escoge por terminal al principio del programa.
// Estos ejemplos se encuentran desarrollados en el archivo signalMessagesExample.c.
// Si se desean probar otros patrones de sennales se pueden crear en dicho archivo.
// Los valores utilizados para sobrepasar rapidamente todos los limites son:
// tciclo = 500, vlimite = 1 y nmaxcic = 3
// Se pueden usar otros valores, no hay problema.
void *signalExamples(void *p){
    switch (example){
        case 1:
        {
            // En este ejemplo se reciben dos sennales de tipo SIGRTMIN
```

```
            // primero y un segundo despues otra sennal de tipo SIGRTMIN
            // y SIGALARM para probar que se apaga el indicador. Si se deja
            // continuar se puede comprobar como se para el programa
            // (porque se cumplen los ciclos maximos sin sennales) que
            // esperara un 0 del terminal de comandos para finalizar completamente.
            testExample1();
            break;
        }
        case 2:
        {
            // Este ejemplo esta pensado para probar los comandos.
            // Se envian constantemente todas las sennales de tipo
            // SIGRTMIN (para N_SIG = 4) cada 500ms.
            while (end!=1){
                testExample2();
            }
            break;
        }
        case 3:
        {
            // Este ejemplo esta vacio. Se parara y esperara como el ejemplo 1.
            testExample3();
            break;
        }
        case 4:
        {
            // Envia un patron de sennales cada segundo.
            // Disennado para probar los comandos y que se vea mejor la activacion
            // y desactivacion de la alarma (porque el segundo ejemplo es demasiado rapido).
            while (end!=1)
            {
                testExample4();
            }
            break;
        }
        case 5:
        {
            // Se envian unas sennales de tipo SIGRTMIN primero y luego se
            // envia la sennal SIGTERM para demostrar que el envio de dicha sennal finaliza el programa.
            // Aun asi ocurre como en los otros casos: se espera que se envie un
            // 0 del terminal de comandos para finalizar completamente.
            testExample5();
        }
    }
}
```

7.2. *commander.c*

```
#include <stdio.h>      // Input/output para sscanf, printf y sprintf.
#include <stdlib.h>     // Macros.
#include <string.h>     // Para tratar los string.
#include <mqueue.h>     // Para las colas de mensajes.
#include <sys/stat.h>   // Macros.
#include "problema.h"   // Cabecera dada por problema.

#define COMMANDER   "/commander-queue"      // Nombre de la cola de comandos (la declaramos aqui para que
                                            // tenga el mismo nombre en el otro programa)
#define MAX_MESSAGES 10                     // Maximo num de mensajes para la cola
#define MAX_MSG_SIZE 256                    // Tamanno maximo de mensaje
#define MSG_BUFFER_SIZE MAX_MSG_SIZE + 10   // Tamanno de mensaje en buffer
                                            // (le sumamos 10 al maximo para darle margen)

int main (int _argc, char **_argv)
{
    mqd_t commander, receiver; // Id de las colas de mensajes.
    printf ("Terminal para envio de comandos al programa principal.\n");

    //Definimos la caracteristicas de la cola.
    struct mq_attr attr;
    attr.mq_flags = 0;
    attr.mq_maxmsg = MAX_MESSAGES;
    attr.mq_msgsize = MAX_MSG_SIZE;
    attr.mq_curmsgs = 0;

    // Abrimos la cola que usaremos aqui que es la que manda los comandos desde el terminal.
    // Se creara la cola si no existe y sino se leera. Ademas damos acceso total de lectura, escritura
    // y ejecucion, y por ultimo le asociamos los atributos anteriores.
    // El uso del if es para deteccion de fallos de comunicacion (para debug).
```

```
    if ((commander = mq_open (COMMANDER, O_RDONLY | O_CREAT, S_IRWXU, &attr)) == -1) {
        perror ("Fallo de terminal de comandos.");
        exit (1);
    }

    char inputBuffer [MSG_BUFFER_SIZE];     // Buffer de entrada para recibir el nombre de la otra cola.
    char outputBuffer [MSG_BUFFER_SIZE];    // Buffer de salida para enviar datos.

    int commandSend = 10000; // Inicializar para que no sea 0.

    printf ("Esperando a programa principal para comenzar.\n");

    while (commandSend!=0) { // El comando 0 finaliza este programa tambien.
        printf("------------------------------------------------------------\n");

        // Recibimos el nombre de la cola que va a recibir los comando.
        // Se hace asi porque el disenno del terminal de comandos es muy parecido
        // al de un servidor que puede recibir distintos clientes y es necesario que
        // cada cliente se ejecute separadamente en cada intercambio de mensajes.
        // El uso del if es para deteccion de fallos de comunicacion (para debug).
        if (mq_receive (commander, inputBuffer, MSG_BUFFER_SIZE, NULL) == -1) {
            perror ("Fallo en programa principal. No se recibe contacto.\n");
            exit (1);
        }

        // Abrimos la cola de mensajes que hemos recibido (el nombre se ha recibido antes en
        // inputbuffer) y lo programamos para poder escribir.
        // El uso del if es para deteccion de fallos de comunicacion (para debug).
        if ((receiver = mq_open (inputBuffer, O_WRONLY)) == 1) {
            perror ("No se consigue abrir la cola del programa principal.\n");
            continue;
        }

        // EL comportamiento de cada comando viene explicado en el codigo de
        // solucion.c en la funcion messageReceiver.
        printf("Escriba el numero del comando que se desee realizar:\n");
        printf("0 = \033[1;34mFIN PROGRAMA\033[0m || 1 = \033[1;34mIGNORAR SEnnAL\033[0m || 2 = \033[1;34
    mCONTAR SEnnAL\033[0m || 3 = \033[1;34mAPAGAR INDICADOR\033[0m\n");
        printf("Comando: ");

        scanf("%d", &commandSend); // Recogemos el valor del comando del terminal.
        sprintf (outputBuffer, "%d", commandSend); // Enviamos el valor del comando al buffer de salida.

        // Enviamos el valor del comando desde el buffer de salida. El tamanno del buffer se define como el
        // tamanno de lo que contiene mas un margen para el caracter de terminacion de string.
        // El uso del if es para deteccion de fallos de comunicacion (para debug).
        if (mq_send (receiver, outputBuffer, strlen(outputBuffer)+1, 0) == -1) {
            perror ("Error al enviar comando.\n");
            continue;
        }

        // Si el comando enviado ha sido el 1 o el 2 se tiene que enviar ademas que sennal
        // es la que se desea modificar el comportamiento.
        if (commandSend == 1 || commandSend == 2){
            int signal;
            printf("Sennal (de 0 a %d): SIGRTMIN + ", N_SIG);
            scanf("%d", &signal); // Lectura de la sennal deseada.
            printf("\n");
            sprintf(outputBuffer, "%d", signal); // Metemos el valor en el buffer de salida para el envio.

            // Envio de la sennal elegida a la cola de mensajes del otro programa.
            // El uso del if es para deteccion de fallos de comunicacion (para debug).
            if (mq_send (receiver, outputBuffer, strlen(outputBuffer)+1, 0) == -1) {
                perror ("Error al enviar comando.\n");
                continue;
            }
        }
        printf ("\033[1;32mComando enviado.\033[0m\n");
    }
}
```