Escuela Técnica Superior de Ingenieros Industriales
Universidad Miguel Hernández

SISTEMAS INFORMÁTICOS DE TIEMPO REAL

2º INDUSTRIALES

"Creación y manejo básico de threads"

Luis Miguel Jiménez
Rafael Puerto Manchón

Departamento de Ingeniería de Sistemas Industriales
Área de Ingeniería de Sistemas y Automática

1.- Introducción

Como se ha visto en teoría, la arquitectura de ejecución de tareas basada en threads ofrece un mecanismo fiable, rápido, sencillo y de bajo coste para programar sistemas concurrentes, y en especial sistemas de tiempo real.

Una de las mayores ventajas de trabajar con threads, en contraposición a procesos, es que distintos threads pueden compartir datos comunes sin tener que recurrir a esquemas de comunicación (memoria compartida, tuberías, etc.). Esto va a permitir desarrollar aplicaciones de una manera más rápida y fácil además de más fiables pues se prescinde de mecanismos complejos para la comunicación y para compartir datos entre distintas tareas.

Otra ventaja importante es la rapidez de su creación y el menor costo que presentan los threads frente a los procesos. Como se vio, existen varios modelos de ejecución de threads: el modelo de threads de usuario, de núcleo e híbrido. Por supuesto el mejor de ellos es el híbrido ya que permite beneficiarse de las ventajas de los dos anteriores. Aún trabajando con un modelo de threads de núcleo (cuya creación es más costosa) se obtienen considerable ventajas frente a la arquitectura de procesos ya que el coste de creación (tanto temporal como de memoria) es inferior al de procesos, al tiempo que permite la comunicación transparente entre threads.

La presente práctica pretende poner de manifiesto estas ventajas mediante la resolución de un problema que, aunque sencillo, podría tratarse de una aplicación real (obviando por supuesto la comunicación con el entorno físico).

2.- Planteamiento del problema

El problema a resolver mediante programación concurrente con threads es el siguiente:

Se dispone de un depósito de agua de una determinada capacidad. Una tubería vierte en el depósito la cantidad de l_1 litros por segundo. Además, el depósito dispone de una válvula de evacuación que permite evacuar l_2 litros cada vez que se abre. Se tiene además un sistema de control *"todo/nada"* (es decir, no actúa sobre el sistema de una forma proporcional, simplemente abre o cierra la válvula de evacuación) que cada 2 segundos mide mediante un sensor el volumen del líquido del depósito en litros. Si el volumen del líquido almacenado en el depósito es mayor que un cierto volumen crítico, el sistema de control abre la válvula de evacuación. En el caso de que el sistema de control no sea capaz de regular el sistema, se dispone de un sistema de alarma que se ejecuta cada segundo y en caso de que la cantidad de líquido almacenada supere un cierto volumen considerado como peligroso, activa una señal de alarma que avisa a un operador para que tome las medidas oportunas (en nuestro caso simplemente activa la señal de alarma). El esquema descrito anteriormente se representa gráficamente en la figura 1.

La resolución del problema consiste en realizar un simulador cuyo comportamiento represente el funcionamiento del sistema descrito anteriormente. Dicho simulador se realizará en C estándar utilizando pthreads (threads POSIX).

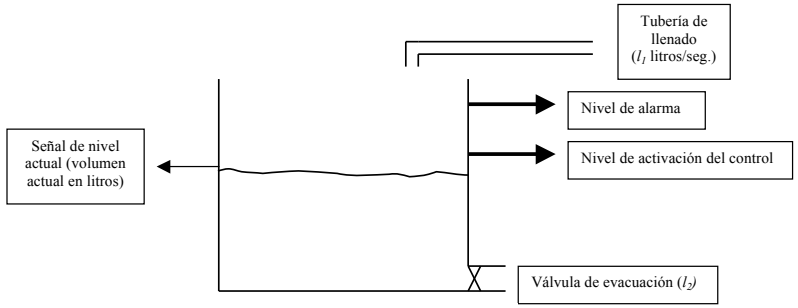

Figura 1. Esquema del sistema

3.- Resolución del problema

Puesto que se pide realizar un simulador del funcionamiento del sistema, en ningún momento se va a disponer de las señales físicas del sistema. Para representar dichas señales se utilizarán variables globales accesibles a todos los threads del programa (con procesos sería imposible utilizar este método de compartir datos). Además, el simulador se ejecutará durante un cierto tiempo denominado *tiempo de simulación*. Puesto que todavía no se ha visto como definir relojes y temporizadores en Unix/POSIX, utilizaremos un bucle que incrementa un entero sin signo hasta que ese entero coincide con otro número que representa el tiempo de simulación.

Como datos de entrada al simulador tendremos:

- Tiempo de simulación (en unidades de tiempo sin definir)
- Volumen en litros/seg. que la tubería de entrada vierte en el depósito
- Volumen en litros que la válvula de evacuación libera cada vez que se abre
- Nivel umbral a partir del cual el controlador debe actuar (en litros)
- Nivel de señal de alarma (en litros)

Veamos a continuación cada uno de los puntos importantes a tener en cuenta en la resolución del problema.

3.1. Variables globales

Como se ha comentado antes, puesto que no podemos disponer de las señales físicas del sistema, representaremos éstas mediante cantidades almacenadas en variables globales. Esto va a permitir que todas las funciones que ejecutan los diferentes threads dispongan de estos valores.

Las variables globales a definir son las siguientes:

- **`volumen`**: variable de tipo entero que representa el volumen actual del depósito en litros.
- **`caudal`**: variable entera que representa el volumen en litros/seg. que la tubería de entrada vierte al depósito.
- **`caudalvaci`**: variable entera que representa el volumen en litros que la válvula de evacuación libera cada vez que se abre.
- **`umbral`**: variable entera que representa el volumen en litros a partir del cual el controlador abre la válvula de evacuación.
- **`peligro`**: variable entera que representa el volumen litros a partir del cual se activa la señal de alarma.
- **`tiempoej`**: variable entera sin signo que representa el tiempo total de ejecución (ver punto 3.2).
- **`tiempo`**: variable entera sin signo que representa el tiempo actual de simulación.

3.2. División de tareas dentro del programa

El simulador se compone de diferentes tareas. Dichas tareas serán realizadas por funciones, donde cada función será ejecutada por un thread diferente.

Las funciones a implementar se definen a continuación:

- **Función** `reloj`: esta función será la encargada de contar el tiempo de simulación. El tiempo total de simulación se debe pedir por teclado (pongamos 10 segundos). Puesto que no disponemos todavía de mecanismos para implementar relojes o temporizadores, se multiplicará el tiempo de ejecución por una cantidad elevada (por ejemplo 20000000), y el valor resultante se almacenará en la variable `tiempoej`. La función reloj simplemente ejecutará un bucle en el que incrementa la variable `tiempo` hasta que sea `tiempoej`. Puesto que la variable `tiempo` es global, todas las demás funciones tienen acceso a ella de forma que comprobarán si el tiempo es menor que `tiempoej` y en ese caso realizarán su tarea. En el caso de que `tiempo` sea igual que `tiempoej`, terminarán. El esquema básico de todas las funciones es el siguiente:

```
void * funcion (void * arg)
{
      while (tiempo<tiempoej)
      {
            // Realizar tarea
      }
      pthread_exit(NULL);
}
```

  Es importante resaltar que el tiempo de ejecución no tiene ninguna correspondencia con ninguna unidad de tiempo, es más, el tiempo de ejecución depende del tiempo que tarde en ejecutarse el bucle en la función `reloj`, y éste depende de la carga actual del sistema por lo que puede ser bastante diferente de unas ejecuciones a otras. Por tanto, este mecanismo nos permite simplemente ejecutar el programa durante "un tiempo".

  Como vemos, todas las funciones devuelven un puntero a `void` y tienen como parámetro un puntero a `void`. En el caso que nos ocupa, puesto que no se pasan argumentos, no es necesario utilizar la variable `arg`. Todas las funciones se definen de la misma forma ya que ésta es la forma en que la función `pthread_create` acepta el parámetro de la función a ejecutar.

- **Función *lanzar***: esta función será la encargada de lanzar los demás threads, es decir, será la única función ejecutada por un thread en la función `main`. Su definición es simple:

```
void * lanzar (void * arg)
{
      pthread_create (&tempo, &attr, reloj, NULL);
      pthread_create (&control, &attr, controldep,NULL);
      pthread_create (&llenar, &attr, llenado, NULL);
      pthread_create (&alar, &attr, alarma, NULL);

      pthread_exit (NULL);
}
```

  Este mecanismo, aunque en este caso no es necesario, permite liberar inmediatamente la función `main`, puesto que después de ser ejecutada esta función (por un thread), la función `main` termina. En otros esquemas de programación con threads, este mecanismo es bastante útil y por eso se ha querido mostrar su uso en este programa.

- **Función *llenado***: esta función realizará la tarea de llenar el depósito en `caudal` litros cada segundo, es decir, sumará `caudal` a `volumen`. Hay que tener en cuenta que esta acción se realiza cada segundo. Para implementar esta acción es necesario realizar la operación de suma y después esperar un segundo (con la función `sleep(s)`) antes de volver a llenar el deposito (sumar). Por supuesto, tal y como se ha comentado antes, todas las funciones se ejecutarán mientras `tiempo` sea menor que `tiempoej`.

- **Función *controldep***: esta función controlará el nivel del depósito mediante la apertura de la válvula de evacuación cuando la variable `volumen` sea mayor que `umbral` (que es el volumen a partir del cual el controlador debe actuar). En caso de que `volumen` sea mayor que `umbral`, esta función resta `caudalvaci` a `volumen`. Esta tarea se realizará cada 2 segundos, tal y como se especifica en el enunciado; por tanto, es necesario esperar 2 segundos antes de realizar una nueva comprobación.

- **Función *alarma***: esta función comprueba cada segundo si `volumen` es mayor que `peligro`, y en caso afirmativo imprimirá un mensaje de alarma indicando el volumen actual del deposito.

3.3. Threads

Cada una de las funciones citadas en el punto anterior deben ser ejecutadas por un thread distinto. Esto va a permitir ejecutar dichas tareas de forma *"pseudoparalela"* dando sensación de concurrencia o paralelismo. Para ello hay que definir un thread por cada función que se ejecute. Así, debemos declarar los threads que se muestran en la tabla 1 asociados a sus correspondientes funciones.

Thread	Función asociada
`start`	`lanzar`
`control`	`controldep`
`llenar`	`llenado`
`alar`	`alarma`
`tempo`	`reloj`
`thmain`	`main`

Tabla 1. Threads y funciones asociadas

Como sabemos, los threads son de tipo `thread_t`. Además, los threads tienen asociados determinados parámetros que son especificados en el instante en que se crean. Estos parámetros se encapsulan en un objeto de tipo `pthread_attr_t` y se deben inicializar por defecto. En nuestro programa definiremos la variable `attr` del tipo anteriormente mencionado para especificar los atributos de los threads en el momento de su creación. Tanto los threads como la variable de atributos deben ser declaradas como variables globales (no es estrictamente necesario pero en este caso otorga más legibilidad al código).

Es importante plantear ahora con que modelo de ejecución de threads debemos trabajar en este programa. Si trabajamos con threads de usuario, los threads no son vistos por el sistema operativo; para el sistema operativo sólo es visible el proceso que los contiene. Aunque este modelo de threads es más rápido que el modelo de threads de núcleo, presenta en esta aplicación un serio inconveniente: si se utilizan llamadas (funciones) que bloquean el proceso (por ejemplo `getchar`, `scanf`, `sleep`,...) todos los threads quedan bloqueados ya que el sistema operativo bloquea todo el proceso. En cambio, si se utiliza un modelo de threads de núcleo, cada thread es una unidad independiente para el sistema operativo y por tanto puede planificar y manejar unas independientemente de otras. De esta forma, si un thread realiza una llamada que lo bloquea, sólo se bloquea el que realiza la llamada, permitiendo a los demás seguir su ejecución normal.

El sistema de pthreads que implementa Sun Solaris crea por defecto todos los threads como threads de usuario, por lo que es necesario cambiar el atributo *contentionscope* del objeto atributo para indicar que los threads que se van a crear deben ser threads de núcleo (ver capítulo 5 de apuntes de teoría).

Por último, como ya se explicó, en esta forma de trabajar con threads es el sistema operativo quien proporciona el mecanismo de gestión de threads. Para ello se dispone de una librería que se ejecuta en tiempo de ejecución y que se encarga de gestionar y planificar los threads. Esta librería se denomina *pthread* y es necesario indicar al compilador que enlace nuestro código objeto con dicha librería. Esto se hace utilizando la opción *–l* de *gcc*. Así, para compilar nuestro programa debemos escribir:

```
SUN-Solaris:      gcc -o prog nombreprog.c -lpthread -lposix4
Linux:            gcc -o prog nombreprog.c -lpthread -lrt
```

donde ***prog*** es el programa ejecutable resultante de la compilación y ***nombreprog.c*** es nuestro fichero de código fuente.

3.4. Esquema del programa

Con el fin de facilitar la realización de la práctica, se incluye a continuación un esquema de cómo debe ser el programa, a la vez que se especifican los ficheros cabecera que debe contener.

```
// Programa que simula el control del volumen de liquido en un
// depósito


#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

// Declaración de prototipos

void * controldep (void *);
void * llenado (void *);
void * alarma  (void *);
void * lanzar (void *);
void * reloj (void *);


// Declaración de threads

pthread_t start, control, llenar, alar, tempo ,thmain;

// Declaración del objeto atributo

pthread_attr_t attr;

// Declaración de variables globales

int volumen, umbral, caudal, caudalvaci, peligro;
unsigned long tiempo, tiempoej;

// Definición de funciones



int main (void)
{
      // Inicialización de variables

      // Inicialización del objeto atributo

      // Cambio del parámetro contentionscope para crear los threads
      // como threads de núcleo

      // Lectura de datos

      tiempoej = tiempoej * 20000000;

      pthread_create(&start, &attr, lanzar, NULL);

      printf("\n Soy main, he lanzado la ejecución y salgo\n");

      pthread_exit(NULL);
}
```

3.5. Ejemplo de ejecución

Con el fin de saber lo que sucede en cada momento, cada función debe escribir un mensaje en pantalla indicando quien es y los parámetros actuales del sistema (volumen, acción que realiza, etc.).

Para los parámetros de entrada que se muestran en la tabla 2 se ha obtenido el resultado que se muestra a continuación.

Parámetro	Variable	Valor
Tiempo de ejecución	tiempoej	100
Volumen entrada	caudal	3 litros/seg.
Volumen salida	caudalvaci	7 litros/seg.
Valor crítico de control	umbral	10 litros
Valor crítico de alarma	peligro	12 litros

Tabla 2. Parámetros de entrada para la simulación de ejemplo

Ejemplo de simulación

Soy main, he lanzado la ejecución y salgo

Llenado: volumen = 3
Llenado: volumen = 6
Llenado: volumen = 9
Llenado: volumen = 12
Control: he reducido el volumen a 5
Llenado: volumen = 8
Llenado: volumen = 11
Control: he reducido el volumen a 4
Llenado: volumen = 7
Llenado: volumen = 10
Llenado: volumen = 13
Alarma: cuidado, el volumen es 13
Llenado: volumen = 16
Alarma: cuidado, el volumen es 16
Control: he reducido el volumen a 9
Llenado: volumen = 12
Llenado: volumen = 15
Alarma: cuidado, el volumen es 15
Control: he reducido el volumen a 8
Llenado: volumen = 11
Llenado: volumen = 14
Alarma: cuidado, el volumen es 14
Control: he reducido el volumen a 7
Llenado: volumen = 10
Llenado: volumen = 13
Alarma: cuidado, el volumen es 13
Control: he reducido el volumen a 6
Llenado: volumen = 9
Llenado: volumen = 12
Control: he reducido el volumen a 5
Llenado: volumen = 8
Llenado: volumen = 11
Control: he reducido el volumen a 4
Llenado: volumen = 7
Llenado: volumen = 10
Llenado: volumen = 13
Alarma: cuidado, el volumen es 13
Llenado: volumen = 16
Alarma: cuidado, el volumen es 16
Control: he reducido el volumen a 9
Llenado: volumen = 12
Llenado: volumen = 15
Alarma: cuidado, el volumen es 15

4.- Ejercicio: secciones críticas, mutex

Los threads que ejecutan cada tarea acceden a variables globales compartidas (***volumen***), para evitar condiciones de carrera, su acceso debe realizarse dentro de una sección crítica. Añadir la sincronización de las secciones críticas mediante mutex..

5.- Ejercicio: uso del reloj de tiempo real

Sustituir el thread ***reloj()*** de forma que realice una medida precisa de tiempo utilizando las funciones de gestión de relojes de tiempo real. Configurar las unidades de las variables tiempo y tiempoej en milisegundos.

Sustituir la función ***sleep()*** por la más precisa ***nanosleep()*** que es no bloqueante con threads de usuario.

Cambiar el modelo de ejecución usando ***Threads de Usuario*** que permiten un cambio de contexto más rápido.

6.- Ejercicio: modificación de los parámetros de planificación

Podemos considerar que la tarea más crítica de las realizadas en el ejercicio anteriores la tarea de alarma. Esto implica que la función alarma debería ejecutarse con más prioridad que las demás tareas con el fin de evitar daños en la planta.

El ejercicio que se propone es cambiar la prioridad del thread que ejecuta dicha tarea a la prioridad más alta de las permitidas en el sistema.