PRÁCTICA 2: CONCURRENCIA E HILOS

v1.3.1

Rodrigo García Carmona

Antes de empezar…

…la práctica, copia los ficheros que se encuentran en la carpeta “SOS-Practica_2-Materiales” al directorio en el que quieras trabajar. Ahora ya estás preparado para comenzar la práctica.

Objetivos de la práctica

Esta práctica presenta tres objetivos básicos. El primero consiste en habituarse a la creación de programas multi-hilo, descubriendo al hacerlo las diferencias existentes entre este tipo de programas y los programas clásicos. Para implementar estos programas se usará el soporte proporcionado por el sistema operativo (utilizando la interfaz POSIX y el lenguaje C).

El segundo objetivo consiste en tratar el problema de la sección crítica mediante un caso práctico de acceso concurrente a recursos (en este caso, variables en memoria) sin mecanismos de protección.

Por último, esta practica finalizará con la programación de un caso completo en el que es necesario asegurar tanto exclusión mutua como sincronización, escrito en C y utilizando o bien semáforos o bien variables condición.

Anexos

Junto a esta práctica se proporcionan 3 anexos, que explican el funcionamiento de las bibliotecas POSIX en lo referente a:

1. Threads.
2. Semáforos.
3. Cerrojos y variables condición.

Échales un ojo antes de seguir con la práctica, y tenlos a mano durante la misma. En ellos se explica de forma resumida cómo se deben usar las bibliotecas POSIX para trabajar con hilos y programas concurrentes.

Los programas de esta práctica son más largos que los del resto, así que asegúrate de que entiendes bien lo que hacen. Analízalos con cuidado y paciencia.

1. Hola Mundo concurrente

El programa “Hola Mundo” es el primer ejemplo que se suele dar para ilustrar un lenguaje de programación. Este programa simplemente escribe por pantalla “Hola Mundo” y acaba.

Utilizaremos este ejemplo clásico como primera toma de contacto de programas concurrentes. Hemos bautizado a este programa, por tanto, como “Hola Mundo concurrente”, y puedes encontrarlo en “hmconc.c”.

hmconc.c:

```
#include <stdio.h>
#include <pthread.h>

/*
    CONSTANTES
    UN_SEGUNDO  : Un segundo expresado en milisegundos
*/

#define UN_SEGUNDO    1000

/*
    TIPOS DE USUARIO
    argumentos_tarea_t : Estructura con los argumentos iniciales de las tareas
*/
typedef struct argumentos_tarea_t {
    int id;     /* Identificador de la tarea */
    int ttl;    /* Time To Live (numero de iteraciones) */
} argumentos_tarea_t;

/*
    FUNCIONES AUXILIARES
    retraso(int milisegunos):       Suspende a la tarea invocante durante
                                    los milisegundos especificados
    imprimir(int id,char *cadena):  Imprime en pantalla la cadena a partir
                                    de la columna id*2
*/

void retraso (int milisegundos) {
    struct timespec ts;
    if (milisegundos > 0) {
        ts.tv_sec  = milisegundos / 1000;
        ts.tv_nsec = (milisegundos % 1000) * 1000000;
        nanosleep (&ts, NULL);
    }
}


void imprimir (int id, char *cadena) {
    int col; // columna en la que tenemos que empezar a imprimir (id*2)
    // Dejamos los espacios indicados al principio:
    for (col = 0; col < (id*2); col = col + 1) {
        fprintf(stderr, " ");
    }
    // Imprimimos el mensaje
    fprintf(stderr, "[Tarea %d]: %s\n\n", id, cadena);
```

```
}

/*
    FUNCIONES QUE EJECUTAN LAS TAREAS
    - Saluda : Nace (saluda), itera y muere
*/
void *saluda (void *argumentos) {
    // Variables locales de la tarea:
    struct argumentos_tarea_t *arg;
    int id;
    int ttl;
    int i = 0;
    // Recuperamos los argumentos con los que se ha creado la tarea
    arg = (argumentos_tarea_t *) argumentos;
    id  = arg->id;
    ttl = arg->ttl;
    // Mensaje de nacimiento
    imprimir(id, "Hoooola mundo, estoy vivo!!");
    // Bucle en el que se espera un poco e imprime
    // (Cada tarea itera tanto como su TTL)
    while (i < ttl) {
        retraso(UN_SEGUNDO);
        imprimir(id, "Sigo viva");
        i = i+1;
    }
    imprimir(id, "Me muero ...");
    pthread_exit(0);
}

/*
    PROGRAMA PRINCIPAL
*/
int main (void) {
    struct argumentos_tarea_t arg1, arg2, arg3;
    pthread_t t_saluda1, t_saluda2, t_saluda3;
    // Primero saludamos (ante todo, educacion!)
    fprintf(stderr, "[PRINCIPAL]: HOLA MUNDO!!\n\n");
    // Preparamos los argumentos iniciales cada tarea y la creamos:
    // (Nota: podriamos haber reutilizado "arg1" en las tres llamadas)
    arg1.id  = 1;
    arg1.ttl = 10;
    pthread_create (&t_saluda1, NULL, saluda, &arg1);
    arg2.id  = 3;
    arg2.ttl = 5;
    pthread_create (&t_saluda2, NULL, saluda, &arg2);
    arg3.id  = 6;
    arg3.ttl = 4;
    pthread_create (&t_saluda3, NULL, saluda, &arg3);
    // Obligamos a que el programa principal se espere a que todas
    // las tareas finalicen
    pthread_join(t_saluda1, NULL);
    pthread_join(t_saluda2, NULL);
```

```
    pthread_join(t_saluda3, NULL);
    // Fin del programa principal
    exit(0);
}
```

Fíjate en que la creación de hilos se realiza explícitamente desde el programa principal. Intenta predecir qué se visualizará en pantalla al ejecutar el programa.

Compila y ejecuta el programa. No olvides incluir el parámetro “-lpthread” en la llamada a “gcc”, tal como se indica en el primer anexo.

Ejercicios

1. ¿Qué función se utiliza para crear cada hilo? ¿Con qué parámetros?
2. ¿Qué consigue el programa principal al realizar un “pthread_join” sobre cada uno de los hilos que ha creado? ¿Qué ocurriría si no hiciera esas llamadas?

3. Concurrencia sin protección

El problema de la sección crítica puede darse en sistemas en los que procesos (o hilos) concurrentes acceden sin mecanismos de protección explícitos a recursos compartidos que sólo pueden ser utilizados en serie (es decir, de uno en uno). En recursos de este tipo se debe garantizar de alguna manera que su acceso por parte de los procesos se serializa completamente, pues de lo contrario se puede incurrir en una inconsistencia del propio recurso. Tal vez el ejemplo más evidente de este problema sean variables en memoria compartidas por varios procesos o hilos. Esta problemática se presenta en esta práctica de forma experimental.

Para ilustrar la necesidad de los mecanismos de protección en estos casos, se han elaborado tres programas muy similares, denominados “sinprot1.c”, “sinprot2.c” y “sinprot3.c”. Los tres programas poseen internamente varios hilos que acceden concurrentemente (y sin protección) a una variable compartida “V”, cuyo valor inicial es 100 en todos los programas. En concreto, cada programa posee:

- Un hilo (denominado “agrega”) que, dentro de un bucle, se dedica a sumar 1 a la variable “V”.
- Un segundo hilo (denominado “resta”) que, dentro de un bucle, va restando 1 a la misma variable.
- Y un tercer hilo (denominado “inspecciona”) que se despierta cada segundo e imprime por pantalla el valor actual de “V”.

Primera versión del programa

Estudia el código del programa “sinprot1.c”. Fíjate sobre todo en el código que ejecutan las funciones “agrega” y “resta”, reproducidas a continuación (el resto del código está en el fichero fuente):

```
void *agrega (void *argumento) {
    long int cont;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        V = V + 1;
        retraso(DORMIR);
    }
```

```
    printf("-------> Fin AGREGA (V = %ld)\n", V);
    pthread_exit(0);
}

void *resta (void *argumento) {
    long int cont;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        V = V - 1;
        retraso(DORMIR);
    }
    printf("-------> Fin RESTA  (V = %ld)\n", V);
    pthread_exit(0);
}
```

De acuerdo con el propio código, y antes de ejecutar el programa, intenta predecir qué se verá por pantalla cuando lo ejecutes. Después, compílalo, ejecútalo, y observa el valor final de “V”

Segunda versión del programa

Estudia el código del programa “sinprot2.c”. Fíjate sobre todo en el código que ejecutan las funciones “agrega” y “resta”, reproducidas a continuación (el resto del código está en el fichero fuente):

```
void *agrega (void *argumento) {
    long int cont;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        V = V + 1;
    }
    printf("-------> Fin AGREGA (V = %ld)\n", V);
    pthread_exit(0);
}

void *resta (void *argumento) {
    long int cont;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        V = V - 1;
    }
    printf("-------> Fin RESTA    (V = %ld)\n", V);
    pthread_exit(0);
}
```

Piensa detenidamente qué diferencias existen entre esta versión y la anterior. De acuerdo con el propio código, y antes de ejecutar el programa, intenta predecir qué se verá por pantalla cuando lo hagas.

Compila y ejecuta el programa y observa lo que se imprime por pantalla, así como el valor final de “V”.

Tercera versión del programa

Estudia el código del programa “sinprot3.c”. Fíjate sobre todo en el código que ejecutan las funciones “agrega” y “resta”, reproducidas a continuación (el resto del código está en el fichero fuente):

```
void *agrega (void *argumento) {
    long int cont;
    long int inter;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        inter = V;
        inter = inter + 1;
        V = inter;
    }
    printf("-------> Fin AGREGA (V = %ld)\n", V);
    pthread_exit(0);
}

void *resta (void *argumento) {
    long int cont;
    long int inter;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        inter = V;
        inter = inter - 1;
        V = inter;
    }
    printf("-------> Fin RESTA    (V = %ld)\n", V);
    pthread_exit(0);
}
```

Piensa detenidamente qué diferencias existen entre esta versión y las anteriores. De acuerdo con el propio código, y antes de ejecutar el programa, intenta predecir qué se verá por pantalla cuando lo hagas.

Compila y ejecuta el programa y observa lo que se imprime por pantalla, así como el valor final de “V”.

Ejercicios

1. Para “sinprot1.c”, ¿cuál es el valor final de la variable “V”? ¿Por qué se da este resultado?
2. Para “sinprot2.c”, ¿cuál es el valor final de la variable “V”? ¿Por qué se da este resultado?
3. Para “sinprot3.c”, ¿cuál es el valor final de la variable “V”? ¿Por qué se da este resultado?

4. Programación con hilos

Ha llegado el momento que utilices las herramientas que tienes a tu disposición para implementar un pequeño problema. Para resolver esta parte de la práctica puedes utilizar semáforos o variables condición (para implementar monitores), como prefieras. En cualquiera de los dos casos, deberás usar C y las bibliotecas POSIX.

Quizá sea un buen momento pare repasar los anexos nombrados al principio de la práctica. Deberás elegir entre usar semáforos o cerrojos y variables condición.

Características del problema

“Les Petits Poissons” es, a pesar de su juventud, una de las peluquerías más famosas de Londres. Esta fama se debe, sin duda alguna, al buen hacer de su dueño, Sir Patrick Fish. Este peluquero, de indudable

talento, es la comidilla de las clases altas británicas, tanto por sus habilidades como por las legendarias historias que se le atribuyen, a cada cual más emocionante e increíble. La más sorprendente de todas ellas es la que le permitió ganar el codiciado título de Sir, entregado por la Reina de Inglaterra en persona. Pero dejemos esa historia para otra ocasión…

Como caso de estudio para el problema de la sincronización entre procesos (o threads), la peluquería “Les Petits Poissons” es un ejemplo excelente. Veámoslo.

La peluquería cuenta con las siguientes características:

- 1 silla de peluquería, en la que Sir Patrick hace su magia.
- 5 cómodos sofás, cada uno con capacidad para 1 persona.
- Una bien dotada barra de bar en la que pueden tomar una copa de pie hasta 15 personas.

Cuando un cliente llega hasta “Les Petits Poissons” actuará según las circunstancias y la buena educación inglesa requieran. Es decir:

1. Si la silla de peluquería está libre se sentará en ella y esperará a que le hagan un corte de pelo que no olvidará.
2. Si la silla está ocupadas pero hay sitio en algún sofá se apalancará cómodamente en él, esperando mientras lee un revista de estilo de vida o escucha el vanguardista hilo musical. Una vez quede libre una de las sillas de peluquería se dirigirá a dicha silla.
3. Si los 5 sofás están ocupados se dirigirá a la barra, dónde esperará tomándose uno de los exóticos cócteles preparados por los excelentes profesionales del local. Una vez quede libre uno de los sillones se dirigirá a dicho sofá.
4. Si la barra está al máximo de su capacidad arqueará las cejas y se marchará por dónde ha venido. La clase alta no espera colas.

Primera mejora

Esta es una mejora opcional. No es necesario hacerla para terminar la práctica, pero sube nota.

Sir Patrick no da a basto, por lo que ha entrenado a dos jóvenes peluqueros que le ayudarán a poder atender a los clientes de Les Petits Poissons. Por tanto, ahora la peluquería cuenta con las siguientes características:

- 3 sillas de peluquería, en las que Sir Patrick y sus dos aprendices hacen su magia.
- 5 cómodos sofás, cada uno con capacidad para 1 persona.
- Una bien dotada barra de bar en la que pueden tomar una copa de pie hasta 15 personas.

Es importante tener en cuenta que Sir Patrick y sus aprendices no cortan el pelo igual de rápido. Consulta los datos numéricos más adelante.

Segunda mejora

Esta es una mejora opcional. No es necesario hacerla para terminar la práctica, pero sube nota. Esta mejora depende de las anteriores.

Tanta fama hace que Sir Patrick esté siempre hasta arriba de trabajo, por lo que no puede permitirse cerrar a ninguna hora del día, y corta el pelo a sus clientes incansablemente, de sol a sol (cómo compagina su

trabajo con sus famosas aventuras nocturas es motivo de habladurías). Por tanto, tanto él como sus dos ayudantes han llegado a la conclusión de que la mejor solución para optimizar su tiempo es dormir en la propia silla en la que cortan el pelo a sus clientes, cosa que harán si ven que no hay nadie esperando en los sofás.

Sobra decir que si un cliente está en posición de que le corten el pelo, se acercará a la silla y emitirá un prácticamente inaudible carraspeo, lo suficientemente alto como para despertar al peluquero (o a uno sus ayudantes), pero lo bastante bajo como para mantener la adecuada compostura.

Tercera mejora

Esta es una mejora opcional. No es necesario hacerla para terminar la práctica, pero sube nota. Esta mejora depende de las anteriores.

¡Es terrible! Sir Patrick se ha dado cuenta de un fallo en su peluquería… ha olvidado cobrar a sus clientes una vez han recibido el corte de pelo. Para paliar tan execrable error decide instalar una máquina registradora que, aunque sólo acepta cheques o tarjetas Visa Oro, necesita de un operador humano para funcionar. Dicho operador será siempre uno de sus ayudantes (¡nunca Sir Patrick!), que atenderán la caja si se encuentran un cliente esperando en ella.

Nótese que los clientes jamás hacen cola (son de clase alta), por lo cual no se levantarán de su silla para acercarse hasta la caja hasta que ésta se encuentre vacía.

Datos numéricos

- *Tiempo necesario para cortar el pelo de Sir Patrick:* de 0 a 400 milisegundos (¡que velocidad!)
- *Tiempo necesario para cortar el pelo de los aprendices:* de 0 a 600 milisegundos (¡que velocidad!)
- *Tiempo necesario para cobrar en la caja:* de 0 a 150 milisegundos
- *Número de clientes para el problema:* 50 clientes
- *Tiempo de llegada de los clientes:* Deberás decidirlo tú a base de hacer experimentos. Tendrá que ser lo bastante bajo como para poner a prueba las capacidades de la peluquería.

Ejercicios

1. Implementa la peluquería “Les Petits Poissons” como un archivo (o archivos) fuente en C. Tendrás que añadir un archivo “leeme.txt” en el que expliques cómo la has implementado.