Sistemas Concurrentes y Distribuidos.
Práctica 1. Sincronización de hebras con semáforos.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 16-17

Índice

Sistemas Concurrentes y Distribuidos.
Práctica 1. Sincronización de hebras con semáforos.

Sección 1
Objetivos

Objetivos.

En esta práctica se realizarán dos implementaciones de dos problemas sencillos de sincronización usando librerías abiertas para programación multihebra y semáforos. Los objetivos son:

- Conocer el *problema del productor-consumidor* y sus aplicaciones.
  - Diseñar una solución al problema basada en semáforos.
  - Implementar esa solución en un programa C/C++ multihebra, usando la funcionalidad de la librería POSIX para:
    - la creación y destrucción de hebras
    - la sincronización de hebras usando semáforos
- Conocer un problema sencillo de sincronización de hebras (el *problema de los fumadores*)
  - Diseñar una solución basada en semáforos, teniendo en cuenta los problemas que pueden aparecer.
  - Implementar la solución a dicho problema en C/C++ usando hebras y semáforos POSIX.

Sección 2
El problema del productor-consumidor

2.1. Descripción del problema.
2.2. Plantillas de código
2.3. Actividades y documentación

Subsección 2.1
Descripción del problema.

Problema y aplicaciones

El problema del productor consumidor surge cuando se quiere diseñar un programa en el cual un proceso o hebra produce items de datos en memoria que otro proceso o hebra consume.

- Un ejemplo sería una aplicación de reproducción de vídeo:
  - El **productor** se encarga de leer de disco o la red y descodificar cada cuadro de vídeo.
  - El **consumidor** lee los cuadros descodificados y los envía a la memoria de vídeo para que se muestren en pantalla

  hay muchos ejemplos de situaciones parecidas.
- En general, el productor calcula o produce una secuencia de items de datos (uno a uno), y el consumidor lee o consume dichos items (tambien uno a uno).
- El tiempo que se tarda en producir un item de datos puede ser variable y en general distinto al que se tarda en consumirlo (también variable).

Solución de dos hebras con un vector de items

Para diseñar un programa que solucione este problema:

- Suele ser conveniente implementar el productor y el consumidor como dos hebras independientes, ya que esto permite tener ocupadas las CPUs disponibles el máximo de tiempo,
- se puede usar una variable compartida que contiene un ítem de datos,
- las esperas asociadas a la lectura y la escritura pueden empeorar la eficiencia. Esto puede mejorarse usando un vector que pueda contener muchos items de datos producidos y pendientes de leer.

Condición de sincronización

En esta situación, la implementación debe asegurar que :

- Cada ítem producido es leído (ningún ítem se pierde)
- Ningún ítem se lee más de una vez.

lo cual implica:

- El productor tendrá que esperar antes de poder escribir en el vector cuando haya creado un ítem pero el vector esté completamente ocupado por ítems pendientes de leer
- El consumidor debe esperar cuando vaya a leer un ítem del vector pero dicho vector no contenga ningún ítem pendiente de leer.
- En algunas aplicaciones el orden de lectura o extracción de datos del buffer debe coincidir con el de escritura o inserción, en otras podría ser irrelevante.

Otras propiedades requeridas

Además de lo anterior:

- El programa **no debe impedir** (mediante los semáforos) que el escritor pueda estar produciendo un item al mismo tiempo que el consumidor esté consumiendo otro item (si se impidiese, no tendría sentido usar programación concurrente para esto).
- Lo anterior se refiere exclusivamente a los subprogramas o funciones encargadas de producir o consumir un dato, no se refiere a las operaciones de extraer un dato del buffer o insertar un dato en dicho buffer.
- En el programa, la producción de un dato y su consumo no emplean mucho tiempo de cálculo, ya que es un ejemplo simplificado. Por tanto, se deben introducir retrasos aleatorios variables para poder experimentar distintos patrones de interfoliación de las hebras (ver las plantillas).

Subsección 2.2
Plantillas de código

Simplificaciones

En esta práctica se diseñará e implementará un ejemplo sencillo en C/C++

- cada ítem de datos será un valor entero de tipo int,
- el orden en el que se leen los items es irrelevante (en principio),
- el productor produce los valores enteros en secuencia, empezando en 1,
- el consumidor escribe cada valor leído en pantalla,
- se usará un vector intermedio de valores tipo int, de tamaño fijo pero arbitrario.

Retraso aleatorio:

Para poder introducir un retraso aleatorio, en el lenguaje C/C++ se puede usar la función **retraso_aleatorio**. Recibe como argumento un número de segundos minimo y otro máximo (pueden tener decimales y ser inferiores a la unidad), y deja la hebra bloqueada durante un intervalo de tiempo aleatorio, distinto cada vez, comprendido entre el mínimo y el máximo:

```
#include <unistd.h> // necesario para usleep()
#include <stdlib.h> // necesario para random(), srandom()
#include <time.h>   // necesario para time()

void retraso_aleatorio( const float smin, const float smax )
{
  static bool primera = true ;
  if ( primera )          // si es la primera vez:
  {  srand(time(NULL)); // inicializar la semilla del generador
     primera = false ;  // no repetir la inicialización
  }
  // calcular un número de segundos aleatorio, entre smin y smax
  const float tsec = smin+(smax-smin)*((float)random()/(float)RAND_MAX);
  // dormir la hebra (los segundos se pasan a microsegundos, multiplicándos por 1 millón)
  usleep( (useconds_t) (tsec*1000000.0)  );
}
```

Funciones para producir y consumir:

Para producir un item de datos, la hebra productora invocará esta función:

```
int producir_dato()
{
    static int contador = 0 ;
    contador = contador + 1 ;
    retraso_aleatorio( 0.1, 0.5 );
    cout << "Productor: dato producido: " << contador << endl ;
    return contador ;
}
```

La hebra consumidora llama a esta otra para consumir un dato:

```
void consumir_dato( int dato )
{
   retraso_aleatorio( 0.1, 0.5 );
   cout << "Consumidor: dato consumido: " << dato << endl ;
}
```

Producir o consumir un dato puede tardar un tiempo aleatorio entre una y cinco décimas de segundo.

Hebras productora y consumidora

Los subprogramas que ejecutan las hebras productora y consumidora son como se indica a continuación (no se incluye la sincronización ni los accesos al vector):

```
void * funcion_productor( void * )
{   for( unsigned i = 0 ; i < num_items ; i++ )
    {   int dato = producir_dato() ;
        // falta: insertar "dato" en el vector
    }
    return NULL ;
}
void * funcion_consumidor( void * )
{   for( unsigned i = 0 ; i < num_items ; i++ )
    {   int dato ;
        // falta: extraer "dato" desde el vector intermedio
        consumir_dato( dato ) ;
    }
    return NULL ;
}
```

Es necesario definir la constante **num_items** con algún valor concreto (entre 50 y 100 es adecuado)

Gestión de la ocupación del vector intermedio

El vector intermedio (*buffer*) tiene una capacidad (número de celdas usables) fija prestablecidada en una constante del programa que llamamos, por ejemplo, **`tam_vec`**.

- La constente **`tam_vec`** deberá ser estrictamente menor que **`num_items`** (entre 10 y 20 sería adecuado).
- En cualquier instante de la ejecución, el número de celdas ocupadas en el vector (por items de datos producidos pero pendientes de leer) es un número entre 0 (el buffer estaría vacío) y **`tam_vec`** (el buffer estaría lleno).
- Además del vector, es necesario usar alguna o algunas variables adicionales que reflejen el estado de ocupación de dicho vector.
- Es necesario estudiar si el acceso a dicha variable o variables **requiere o no requiere sincronización alguna** entre el productor y el consumidor.

Soluciones para la gestión de la ocupación

Hay básicamente dos alternativas posibles para gestionar la ocupación, se detallan aquí:

- **LIFO** (pila acotada), se usa una única variable entera no negativa:
  - **`primera_libre`** = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir, y se decrementa al leer.
- **FIFO** (cola circular), se usan dos variables enteras no negativas:
  - **`primera_ocupada`** = índice en el vector de la primera celda ocupada (inicialmente `0`). Esta variable se incrementa al leer (módulo **`tam_vec`**).
  - **`primera_libre`** = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir (módulo **`tam_vec`**).

(asumismos que los índices del vector van desde `0` hasta **`tam_vec`**`-1`, ambos incluidos)

Seguimiento de las operaciones sobre el buffer

Para poder depurar el programa y hacer seguimiento:

- Es conveniente incluir sentencias para imprimir en pantalla el valor insertado o extraido del buffer, justo después de cada vez que se hace (estos mensajes son **adicionales** a los mensjes. que se imprimen al producir o al consumir). Por tanto, se dan dos pasos:
  1. Insertar o extraer el dato del buffer
  2. Escribir un mensaje en pantalla indicando lo que se ha hecho
- Ten en cuenta que el estado de la simulación puede cambiar (y en pantalla pueden aparecer otros mensajes) después del paso 1, pero antes del paso 2.
- Esto puede llevar a cierta confusión, ya que los mensajes no reflejan el estado cuando aparecen, sino un estado antiguo distinto del actual.

Diseña e implementa una solución a este problema, de forma que los mensajes reflejen correctamente el estado en el momento que aparecen.

Subsección 2.3
Actividades y documentación

Lista de actividades

Debes realizar las siguientes actividades en el orden indicado:

1. Diseña una solución que permita conocer qué entradas del vector están ocupadas y qué entradas están libres (usa alguna de las dos opciones dadas).
2. Diseña una solución, mediante semáforos, que permita realizar las esperas necesarias para cumplir los requisitos descritos.
3. Implementa la solución descrita en un programa C/C++ con hebras y semáforos POSIX, completando las plantillas incluidas en este guión. Ten en cuenta que el programa debe escribir la palabra **`fin`** cuando hayan terminado las dos hebras.
4. Comprueba que tu programa es correcto: verifica que cada número natural producido es consumido exactamente una vez.

Documentación a incluir dentro del portafolios

Se incorporará al portafolios un documento indicando la siguiente información:

1. Describe la variable o variables necesarias, y cómo se determina en qué posición se puede escribir y en qué posición se puede leer.
2. Describe los semáforos necesarios, la utilidad de los mismos, el valor inicial y en qué puntos del programa se debe usar **sem_wait** y **sem_signal** sobre ellos.
3. Incluye el código fuente completo de la solución adoptada.

Sección 3
El problema de los fumadores.

3.1. Descripción del problema.
3.2. Plantillas de código
3.3. Actividades y documentación.

Subsección 3.1
Descripción del problema.

Descripción del problema (1)

En este apartado se intenta resolver un problema algo más complejo usando hebras y semáforos POSIX.
Considerar un estanco en el que hay tres fumadores y un estanquero.

1.1. Cada fumador representa una hebra que realiza una actividad (fumar), invocando a una función **fumar**, en un bucle infinito.

1.2. Cada fumador debe esperar antes de fumar a que se den ciertas condiciones (tener suministros para fumar), que dependen de la actividad del proceso que representa al estanquero.

1.3. El estanquero produce suministros para que los fumadores puedan fumar, también en un bucle infinito.

1.4. Para asegurar concurrencia real, es importante tener en cuenta que la solución diseñada **debe permitir que varios fumadores fumen simultáneamente**.

Descripción del problema (2)

A continuación se describen los requisitos para que los fumadores puedan fumar y el funcionamiento del proceso estanquero:

2.1. Antes de fumar es necesario liar un cigarro, para ello el fumador necesita tres ingredientes: tabaco, papel y cerillas.

2.2. Uno de los fumadores tiene papel y tabaco, otro tiene papel y cerillas, y otro tabaco y cerillas.

2.3. El estanquero selecciona **aleatoriamente un ingrediente** de los tres que se necesitan para hacer un cigarro, lo pone en el mostrador, desbloquea al fumador que necesita dicho ingrediente y después se bloquea, esperando la retirada del ingrediente.

2.4. El fumador desbloqueado toma el ingrediente del mostrador, desbloquea al estanquero para que pueda seguir sirviendo ingredientes y **después** fuma durante un tiempo aleatorio.

2.5. El estanquero, cuando se desbloquea, vuelve a poner un ingrediente aleatorio en el mostrador, y se repite el ciclo.

Subsección 3.2
Plantillas de código

Simulación de la acción de fumar

Para simular la acción de fumar se puede usar la función **fumar**, que tiene como parámetro el número de fumador, e invoca a **retraso_aleatorio**, que ya se introdujo antes. El esquema del programa es como sigue:

```
// función que simula la acción de fumar, como un retardo aleatorio de la hebra.
// recibe como parámetro el numero de fumador
// el tiempo que tarda en fumar está entre dos y ocho décimas de segundo.

void fumar( int num_fum )
{
  cout << "Fumador número " << num_fum << ": comienza a fumar." <<endl;
  retraso_aleatorio( 0.2, 0.8 );
  cout << "Fumador número " << num_fum << ": termina de fumar." <<endl;
}

void * funcion_estanquero( void * ) { .... }
void * funcion_fumador( void * num_fum_void ) { .... }

int main()
{
  srand( time(NULL) ); // inicializa la semilla aleatoria
  // ....crear semáforos y poner en marcha hebras...
}
```

Subsección 3.3
Actividades y documentación.

Diseño de la solución

Diseña e implementa una solución al problema en C/C++ usando cuatro hebras y los semáforos necesarios. La solución debe cumplir los requisitos incluidos en la descripción, y además debe:

- Evitar interbloqueos entre las distintas hebras.
- Producir mensajes en la salida estándar que permitan hacer un seguimiento de la actividad de las hebras:
  - El estanquero debe indicar cuándo produce un suministro y qué suministro produce. Para establecer el ingrediente concreto (o, lo que es equivalente, directamente el fumador que podría usarlo), se debe usar también la función **rand()**.
  - Cada fumador debe indicar cuándo espera, qué producto espera, y cuándo comienza y finaliza de fumar.

Documentación a incluir dentro del portafolios

Se incorporará al portafolios un documento incluyendo los siguientes puntos:

1. Semáforos necesarios para sincronización y para cada uno de ellos:
   - Utilidad.
   - Valor inicial.
   - Hebras que hacen **sem_wait** y **sem_signal** sobre dicho semáforo.
2. Código fuente completo de la solución adoptada.