



UNIVERSIDAD DE GRANADA

SISTEMAS CONCURRENTES Y DISTRIBUIDOS

Portafolios de prácticas

Contiene las prácticas del curso

Javier Sáez de la Coba

2º A2

Curso 2017-2018

23 de octubre de 2017

Índice

1. Seminario 1	2
1.1. Cálculo de una integral usando programación concurrente	2
2. Práctica 1	3
2.1. Problema del productor consumidor	3
2.2. Problema de los fumadores	4
A. Entorno de ejecución	7

1. Seminario 1

1.1. Cálculo de una integral usando programación concurrente

Se ha optado por la solución contigua, ya que simplifica los cálculos y la implementación del for que corre en cada hebra.

El valor resultado es:

```
g++ -pthread ejemplo09-plantilla.cpp -o ejemplo9 --std=c++11
javier@javier-zenbook:~/D/2/S/Seminario1
./ejemplo9
Número de muestras (m)      : 1073741824
Número de hebras (n)       : 4
Valor de PI                  : 3.14159265358979312
Resultado secuencial         : 3.14159265358998185
Resultado concurrente        : 3.14159265172718216
Tiempo secuencial           : 19261 milisegundos.
Tiempo concurrente          : 5434.5 milisegundos.
Porcentaje t.conc/t.sec.    : 28.21%
```

Se puede apreciar la correlación entre el número de hebras y la mejora del tiempo de ejecución.

El código que ha generado este resultado es el siguiente:

```
double funcion_hebra( long id )
{
    long inferior = (m*(id-1))/n +1;
    long superior = (m*(id))/n;
    double semisuma = 0.0;
    for (long i = inferior; i < superior; i++) {
        semisuma += f( (i+double(0.5)) /m );
    }
    return semisuma;
}
```

```
double calcular_integral_concurrente( )
{
    future<double> hebras[n];
    double suma = 0.0;
    for (int i = 0; i < n; i++) {
```

```

        hebras[i] = async( launch::async, funcion_hebra, i+1);
    }

    for (int i = 0; i < n; i++) {
        suma += hebras[i].get();
    }

    return suma/m;
}

```

2. Práctica 1

2.1. Problema del productor consumidor

En este apartado vamos a resolver los ejercicios relacionados con el problema del productor consumidor (versión LIFO). En esta versión de la solución, se utiliza un único buffer que es leído como una pila. De ese modo, el último elemento producido e insertado en el buffer es el primero que es leído.

Se ha intentado implementar la solución FIFO pero no ha sido posible debido a errores en el uso del buffer circular.

Para la solución implementada, necesitamos varias variables compartidas: dos semáforos (uno de lectura y otro de escritura), el propio vector de elementos que actúa de buffer de lectura/escritura y una variable de tipo entero que indica cual es la primera posición libre en el vector usado.

```

int buffer[tam_vec-1]; //Vector de elementos producidos
int primera_libre = 0; //Indice de posición libre del vector
Semaphore libres = tam_vec; //Semáforo que bloquea la escritura
Semaphore ocupadas = 0; //Semáforo que bloquea la lectura

```

Para determinar la posición en la que leer o escribir se hace la siguiente operación:

- La hebra productora escribe en la posición guardada por `primera_libre`
- La hebra productora lee el buffer en la posición determinada por `primera_libre - 1`

Así mismo, los dos semáforos: `libres` y `ocupadas` tienen los valores iniciales `tam_vec` y `0` respectivamente. De este modo, la hebra productora consulta el semáforo `libres` que tiene

como valor el número de casillas libres del vector. En caso de que estuviera lleno, el valor del semáforo sería 0 y bloquearía la hebra, que se desbloquearía cuando la hebra consumidora lea un valor del vector y haga un `sem_signal` sobre el semáforo.

De manera similar funciona el semáforo ocupadas, cuyo valor corresponde con los datos producidos y listos para consumir del vector. De manera que la función de la hebra consumidora hace un `sem_wait` sobre el semáforo de casillas ocupadas a la espera que la hebra productora introduzca algún valor.

A continuación se presenta el código de las hebras productoras y consumidoras.

```
void funcion_hebra_productora ( )
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int dato = producir_dato ( ) ;
        sem_wait( libres );
        buffer[ primera_libre ] = dato ;
        primera_libre ++ ;
        sem_signal( ocupadas );
    }
}
```

```
void funcion_hebra_consumidora ( )
{
    for( unsigned i = 0 ; i < num_items ; i++ )
    {
        int dato ;
        sem_wait( ocupadas );
        dato = buffer[ primera_libre - 1 ];
        primera_libre -- ;
        sem_signal( libres );
        consumir_dato( dato ) ;
    }
}
```

2.2. Problema de los fumadores

Para la solución del problema de los fumadores (variante del productor-consumidor) se ha recurrido a los siguientes semáforos:

- `mostrador_vacio`. Como únicamente puede haber un ingrediente en el mostrador y el estantero no puede poner otro hasta que un fumador utiliza el producido se usa un

semáforo cuyo valor inicial es 1 y sobre el que la hebra del estancero hace un `sem_wait`. Las hebras de los fumadores son las que hacen un `sem_signal` sobre el semáforo.

- **ingredientes:** Es un array de semáforos, uno por cada ingrediente posible (y por tanto, según las condiciones del problema, por cada fumador. Especifica que ingredientes hay disponibles. Así cada fumador que necesite un ingrediente específico, hace un `sem_wait` sobre el semáforo de dicho ingrediente y se pone a la cola de espera. El estancero hace un `sem_signal` sobre el semáforo del ingrediente que acaba de producir (y poner en el mostrador) en cada iteración.

Las variables compartidas implementadas han sido:

```
Semaphore ingredientes [3] = {0,0,0};  
Semaphore mostrador_vacio = 1; //El mostrador empieza vacío
```

Las distintas funciones implementadas han sido las siguientes:

Hebra del estancero:

```
void funcion_hebra_estancero ( )  
{  
    while ( true ) {  
        int ingrediente = Producir();  
        sem_wait( mostrador_vacio );  
        cout << "Puesto ingrediente " << ingrediente << endl;  
        sem_signal( ingredientes [ingrediente] );  
    }  
}
```

Hebra del fumador:

```
void funcion_hebra_fumador( int num_fumador )  
{  
    while( true )  
    {  
        sem_wait( ingredientes [num_fumador] );  
        cout << "          Retirado ingrediente " << num_fumador <<  
            endl;  
        sem_signal( mostrador_vacio );  
        fumar( num_fumador );  
    }  
}
```

Función de producir un ingrediente:

```
int Producir() {  
    // calcular milisegundos aleatorios de duración de producir  
    chrono::milliseconds dur_producir( aleatorio<20,200>() );  
    this_thread::sleep_for( dur_producir ); //Esperar  
    int resultado = aleatorio<0,2>();  
    cout << "Estanquero produce ingrediente " << resultado << endl;  
    return resultado;  
}
```

A. Entorno de ejecución

javier@javier-zenbook
OS: Ubuntu 16.04 xenial
Kernel: x86_64 Linux 4.10.0-35-lowlatency
Uptime: 3d 4h 14m
Packages: 3866
Shell: fish
Resolution: 1920x1080
DE: Cinnamon 2.8.6
WM: Muffin
WM Theme: (Numix)
Adwaita [GTK2/3]
Icon Theme: ubuntustudio
Font: Sans 9
CPU: Intel Core i7-7500U CPU @ 3.5GHz
GPU: Mesa DRI Intel(R) HD Graphics 620 (Kabylake GT2)
RAM: 2850MiB / 7862MiB