



UNIVERSIDAD  
DE GRANADA

# Sistemas Concurrentes y Distribuidos:

## Práctica 1. Sincronización de hebras con semáforos.

---

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera  
2022-23

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.  
Dpt. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

## Práctica 1. Sincronización de hebras con semáforos.

### Índice.

1. Objetivos. Espera bloqueada.
2. El problema del productor-consumidor
3. El problema de los (múltiples) productores y consumidores
4. El problema de los fumadores.

## Sección 1.

### Objetivos. Espera bloqueada..

# 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 como se pueden generar números aleatorios y dejar una hebra bloqueada durante un intervalo de tiempo finito.
- ▶ Conocer el *problema del productor-consumidor* y sus aplicaciones.
  - ▶ Diseñar una solución al problema basada en semáforos.
  - ▶ Implementar esa solución con la biblioteca para 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 esa solución con la biblioteca para semáforos.

# Generación de números aleatorios

En C++11 se ofrecen distintas clases para generar números aleatorios.

- ▶ Por simplicidad, usaremos la plantilla de función de nombre **aleatorio**, ya implementada en **scd.h**
- ▶ Sirve para generar un número entero aleatorio, cuyo valor estará entre un mínimo y un máximo (ambos incluidos), deben ser dos constantes conocidas al compilar, p.ej:

```
const int desde = 34, hasta = 45 ; // const es necesario (o constexpr)
....
num1 = aleatorio< 0, 2 >();        // núm. aleatorio entre 0 y 2
num2 = aleatorio< desde, hasta >(); // aleatorio entre 34 y 45
num3 = aleatorio< desde, 65 >();   // aleatorio entre 34 y 65
```

# Espera bloqueada de una hebra

En los ejemplos de esta prácticas queremos introducir esperas bloqueadas en las hebras

- ▶ El objetivo es simular la realización de trabajo útil por parte de las hebras durante un intervalo de tiempo.
- ▶ Las esperas serán de una duración aleatoria, para producir una variedad mayor de posibles interfoliaciones.

Para hacer las esperas se puede usar el método **sleep\_for** de la clase **this\_thread**. El argumento es un valor de tipo **duration**. En este ejemplo usamos una duración en milisegundos (milésimas de segundo):

```
// calcular una duración aleatoria de entre 20 y 200 milisegundos
chrono::milliseconds duracion_bloqueo_ms( aleatorio<20,200>() );
// esperar durante ese tiempo
this_thread::sleep_for( duracion_bloqueo_ms );
```

## Sección 2.

### El problema del productor-consumidor.

- 2.1. Descripción del problema.
- 2.2. Diseño de la sincronización con semáforos
- 2.3. Plantillas para la implementación
- 2.4. Actividades y documentación

Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 2. El problema del productor-consumidor

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 decodificar cada cuadro de vídeo.
  - ▶ El **consumidor** lee los cuadros decodificados 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, pero 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.
- ▶ Para ello, usamos un vector o array de tamaño fijo conocido  $k$ .

# Esquema de las hebras sin sincronización

La hebra productora y la consumidora ejecutan ambas un bucle. La productora produce e inserta un valor en el buffer intermedio. La consumidora extrae un valor un los consume. El esquema (en pseudo-código) puede ser así:

```
{ variables compartidas y valores iniciales }  
var tam_vec    : integer := k ;           { tamaño del vector    }  
    num_items  : integer := .... ;        { número de items      }  
    vec        : array[0..tam_vec-1] of integer; { vector intermedio  }
```

```
process HebraProductora ;  
var a : integer ;  
begin  
  for i := 0 to num_items-1 do begin  
    a := ProducirValor() ;  
    { Sentencia E:                }  
    { (insertar valor 'a' en 'vec') }  
  end  
end
```

```
process HebraConsumidora  
var b : integer ;  
begin  
  for i := 0 to num_items-1 do begin  
    { Sentencia L:                }  
    { (extraer valor 'b' de 'vec') }  
    ConsumirValor(b) ;  
  end  
end
```



# 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).

Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 2. El problema del productor-consumidor

Subsección 2.2.

**Diseño de la sincronización con semáforos.**

# Condiciones de sincronización

Durante la ejecución, en cualquier estado,

- ▶ El total de valores insertados en el buffer desde el inicio es  $\#E$ .
- ▶ El total de valores extraídos desde el inicio es  $\#L$ .
- ▶ El número de valores insertados en el buffer, y todavía pendientes de ser extraídos es  $\#E - \#L$  (lo llamamos *ocupación* del buffer)

Por tanto, surgen estas dos condiciones de sincronización:

- ▶ En ningún momento pueden haberse extraído más valores de los que se han insertado, es decir:

$$\#L \leq \#E$$

- ▶ En ningún momento la ocupación del buffer puede ser superior a su tamaño fijo conocido ( $k$ ), es decir:

$$\#E - \#L \leq k$$

# Diseño de los semáforos

Por tanto, las dos condiciones anteriores pueden escribirse de esta forma:

$$0 \leq \#E - \#L \quad \text{and} \quad 0 \leq k + \#L - \#E$$

Y, al igual que en otros ejemplos, podemos sincronizar los procesos con dos semáforos:

- Un semáforo llamado **ocupadas**, cuyo valor será

$$0 \leq \#E - \#L$$

*Sem\_wait antes de L (signo -)  
Sem\_signal después de E (signo +)*

(es el número de entradas ocupadas del buffer, inicialmente 0)

- Un semáforo llamado **libres**, cuyo valor será

$$0 \leq k + \#L - \#E$$



*Sem\_wait antes de E (signo -)  
Sem\_signal después de L (signo +)*


(es el número de entradas libres del buffer, inicialmente k).





# Esquema de las hebras con sincronización


Ahora podemos incluir las declaraciones de los semáforos y las correspondientes operaciones:

```
{ variables compartidas y valores iniciales }
var tam_vec      : integer := .... ;           { tamaño del vector      }
    num_items    : integer := .... ;           { número de items        }
    vec          : array[0..tam_vec-1] of integer; { vector intermedio      }
    libres  semaphore := tam_vec; { núm. entradas libres (k + #L - #E) }
    ocupadas  semaphore := 0 ;           { núm. entradas ocup. (#E - #L) }
```

```
process HebraProductora ;
var a : integer ;
begin
  for i := 0 to num_items-1 do begin
    a := ProducirValor() ;
    sem_wait( libres  (L-1) )
    { Sentencia E: }
    { (insertar valor 'a' en 'vec') }
    sem_signal( ocupadas ); (ocup. +1)
  end
end
```



```
process HebraConsumidora
var b : integer ;
begin
  for i := 0 to num_items-1 do begin
    sem_wait( ocupadas  (ocup. -1) )
    { Sentencia L: }
    { (extraer valor 'b' de 'vec') }
    sem_signal( libres ); (libres + 1)
    ConsumirValor(b) ;
  end
end
```



Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 2. El problema del productor-consumidor

Subsección 2.3.

**Plantillas para la implementación.**

# Características

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, **empezando en 0, no 1.**
- ▶ El consumidor escribe cada valor leído en pantalla,
- ▶ Se usará un array compartido de valores tipo **`int`**, de tamaño fijo pero arbitrario.

# Funciones para producir y consumir:

La hebra productora llama a **producir\_dato** para producir el siguiente dato, incluye un retraso aleatorio y usa la variable global **siguiente\_dato** (inicializada a 0):

2ª vuelta

```
unsigned producir_dato()
{
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
    const unsigned dato_producido = siguiente_dato ;--> es como el contador
    siguiente_dato++ ; // incrementarlo para la próxima llamada
    cont_prod[dato_producido] ++ ; // incrementar contador de verificación
    cout << "producido: " << dato_producido << endl ;
    return dato_producido ;
}
```

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

```
void consumir_dato( int dato )
{
    this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
    cout << "Consumidor: dato consumido: " << dato << endl ;
}
```

# Funciones de las hebras productora y consumidora

Las funciones que ejecutan las hebras tienen esta forma:

```
void funcion_hebra_productora( )
{  for( unsigned i = 0 ; i < num_items ; i++ )
    {  unsigned dato = producir_dato() ;
        // falta aquí: insertar dato en el vector intermedio:
        // .....
    }
}

void funcion_hebra_consumidora( )
{  for( unsigned i = 0 ; i < num_items ; i++ )
    {  unsigned dato ;
        // falta aquí: extraer dato desde el vector intermedio
        // .....
        consumir_dato( dato ) ;
    }
}
```

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*) será un array C++ de tamaño fijo, tiene una capacidad (número de celdas usables) fija preestablecida en una constante del programa que llamamos, por ejemplo, `tam_vec` (contiene el valor  $k$ )

- ▶ La constante `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 (1/2)

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

- ▶ LIFO (pila acotada), se usa una variable entera ( $\geq 0$ ):
  - ▶ **primera\_libre** = índice en el vector de la primera celda libre (inicialmente 0). Esta variable se incrementa al escribir, y se decrementa al leer. Esta variable necesita exclusión mutua
- ▶ 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`).

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

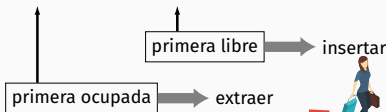
# Soluciones para la gestión de la ocupación (2/2)

En un momento dado, suponiendo ( $k = \text{tam\_vec} = 8$ ), el estado del vector puede ser como se ve aquí. Las celdas ocupadas (en gris) contienen valores enteros pendientes de leer:

**LIFO** (pila acotada: último en entrar es el primero en salir)



**FIFO** (cola circular: primero en entrar es el primero en salir)





# 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 extraído del buffer, justo después de cada vez que se hace (estos mensajes son **adicionales** a los mensajes. 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.

1ª iteración

HEBRA 1  
productora

for (i=0; i < num\_iters; ...)

dato = producir\_dato();

Cont[0]++; → ~~0~~ 1 2 3  
return índice dato → dato = 0

SEM\_WAIT (libres); → libres --;

Excepción  
AUTUA (SC)

buffer[0] = 0;  
primera\_libre ++;

SEM\_SIGNAL (ocupadas); → ocupadas ++;

BUFFER  
0  
primera\_libre

BUFFER  
primera\_libre

HEBRA 2  
consumidora

for (i=0; i < num\_iters; ...)

SEM\_WAIT (ocupadas)

dato = buffer[0];  
primera\_libre --;

Excepción  
AUTUA (SC)

SEM\_SIGNAL (libres);

Consumir\_dato (dato);

consumir\_dato[datos=0]++; → ~~0~~ 1 2 3

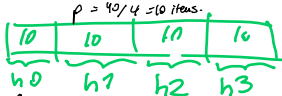
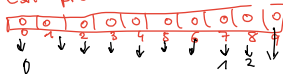
cont ← consumido = índice 0

PRODUCIDORA (10)

cont - prod



Cont - prod - hebra



$$\begin{aligned}
 h_0 &: \{ i \cdot p, i \cdot p + (p-1) \} \\
 h_0 &: \{ 0 \cdot 10, 0 + (9) \} \quad 10 \\
 h_1 &: \{ 10, 10 + 9 \} \quad 10 \\
 h_2 &: \{ 20, 20 + 9 \} \quad 10 \\
 h_3 &: \{ 30, 30 + 9 \} \quad 10
 \end{aligned}$$

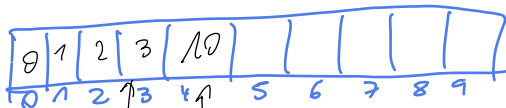
CONSUMIDORA (10)

cont - cons



espera a poder consumir ...

v. intermedio



$h_5$  consume 4  
 $h_6$  produce 10

Con sensor  
libres  
Controlar que  
no se supere  
el tamaño del  
buffer al  
leer o  
meter.

Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 2. El problema del productor-consumidor

Subsección 2.4.

**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 C++11 y usando la biblioteca de semáforos, 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 (múltiples) productores y consumidores.

# Múltiples productores y consumidores

En este ejercicio queremos extender la solución al problema del productor-consumidor, manteniendo todos los requerimientos del mismo, pero ahora:

- ▶ Queremos permitir un número de hebras productoras que no tiene que ser necesariamente la unidad (puede ser mayor).
- ▶ Igualmente podemos tener un número de hebras consumidoras superior a uno (no necesariamente igual al número de productoras) **lo único que no se puede hacer a la vez es tocar el vector**
- ▶ Varios productores pueden estar produciendo a la vez, y varios consumidores pueden estar consumiendo a la vez.
- ▶ La gestión del vector es similar a antes (haz una opción FIFO y una LIFO).



# Actividad: múltiples productores y consumidores

Copia el archivo `prodcons.cpp` en `prodcons-multi.cpp`, y en este nuevo archivo adapta la implementación para permitir múltiples productores y consumidores.

Ten en cuenta estos requerimientos:

- ▶ El número de hebras productoras es una constante  $n_p$ , ( $> 0$ ). El número de hebras consumidoras será otra constante  $n_c$  ( $> 0$ ). Ambos valores deben ser divisores del número de items a producir  $m$ , y no tienen que ser necesariamente iguales. Se definen en el programa como dos constantes (con nombres descriptivos).
- ▶ Cada productor produce  $p = m/n_p$  items. Cada consumidor consume  $c = m/n_c$  items.
- ▶ Cada entero entre 0 y  $m - 1$  es producido una única vez (igual que antes).

# Diseño de la solución: hebras y producción

Para poder cumplir los requisitos anteriores:

- ▶ Las funciones **producir\_dato** y **consumir\_dato** tienen ahora como argumento el número de hebra productora (o consumidora) que lo invoca (un valor  $i$  entre 0 y  $n_p - 1$  (o entre 0 y  $n_c - 1$ ), ambos incluidos).
- ▶ La hebra productora número  $i$  produce de forma consecutiva los  $p$  números enteros que hay entre el número  $i \cdot p$  y el número  $i \cdot p + (p - 1)$ , ambos incluidos.
- ▶ Para esto, debemos tener un array compartido con  $n_p$  entradas que indique, en cada momento, para cada hebra productora, cuantos items ha producido ya. Este array se consulta y actualiza en **producir\_dato**. Debe estar inicializado a 0. La hebra productora  $i$  es la única que usa la entrada número  $i$  (por tanto no hay requerimientos de EM en los accesos a este array).

# Diseño de la solución: concurrencia

nos olvidamos de los mutex, y como e.m. sem=0 inicializado

La solución debe solucionar los problemas de exclusión mutua **exclusivamente usando semáforos**, pero también debe permitir el mayor grado de paralelismo potencial posible. Ten en cuenta:

- ▶ En la solución LIFO:
  - ▶ Dos o más hebras pueden intentar a la vez leer y modificar la variable **primera\_libre** y la correspondiente celda del vector.  
necesitamos semáforo porque tenemos varios
- ▶ En la solución FIFO: **productores accediendo a p\_libre y varios a p\_ocupada**
  - ▶ Dos o más hebras productoras pueden intentar leer y modificar a la vez la variable **primera\_libre** y la correspondiente celda.
  - ▶ Dos o más hebras consumidoras pueden intentar leer y modificar a la vez la variable **primera\_ocupada** y la correspondiente celda.

## Sección 4.

### El problema de los fumadores..

- 4.1. Descripción del problema.
- 4.2. Plantillas de código
- 4.3. Actividades y documentación.

Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 4. El problema de los fumadores.

Subsección 4.1.

**Descripción del problema..**

# Descripción del problema (1)

En este apartado se intenta resolver un problema algo más complejo usando hebras y semáforos. Considerar un estanco en el que hay tres fumadores y un estancuero. Se deben tener en cuenta estos requisitos:

- 1.1. Cada fumador representa una hebra que realiza una actividad (fumar), invocando a una función <sup>(mensaje + retraso)</sup> **fumar**, en un bucle infinito.
- 1.2. Cada fumador debe esperar antes de fumar a que se den ciertas condiciones <sup>(tener ingrediente)</sup> (tener suministros para fumar), que dependen de la actividad del proceso que representa al estancuero.
- 1.3. El estancuero produce suministros para que los fumadores puedan fumar, también en un bucle infinito. <sup>Produce ingrediente aleatorio (solo 1 ítem del ingrediente)</sup>
- 1.4. Para asegurar concurrencia real, es importante tener en cuenta que la solución diseñada **debe permitir que varios fumadores fumen simultáneamente.**

*Fum() necesita ítem del ingrediente 0      ANTES DE FUMAR*

## Descripción del problema (2)

*Cuando el estancero produce debe despertar a fumador.*

Además de los anteriores requisitos, se deben tener en cuenta estos:

- 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 estancero 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 estancero para que pueda seguir sirviendo ingredientes y **después** fuma durante un tiempo aleatorio.
- 2.5. El estancero, cuando se desbloquea, vuelve a poner un ingrediente aleatorio en el mostrador, *y espera a que llegue alguien* y se repite el ciclo.

# Sentencias y funciones

Para poder expresar las condiciones de sincronización fácilmente, haremos estos supuestos:

- ▶ Numeramos los fumadores como fumadores 0,1 y 2. Se numeran igualmente los ingredientes. El fumador número  $i$  necesita obtener el ingrediente número  $i$  para fumar.
- ▶ Llamamos  $P_i$  a una sentencia que ejecuta el estanquero cuando pone el ingrediente número  $i$  en el mostrador. Consiste en la impresión de un mensaje informativo (tipo: “*estanquero produce ingrediente i*”). **poner el ingrediente i en el mostrador**
- ▶ Llamamos  $R_i$  a una sentencia que ejecuta el fumador número  $i$ , por la cual retira el ingrediente  $i$  del mostrador, previamente a fumar. Consiste en imprimir un mensaje informativo (tipo “*el fumador i retira su ingrediente*”) **retirar el ingrediente i del mostrador**



# Esquema de las hebras sin sincronización

El esquema de las hebras es como sigue:

```
process HebraEstanquero ;  
var i : integer ;  
begin  
  while true do begin  
    { simular la producción: }  
    i := Producir() ;  
    { Sentencia  $P_i$  : }  
    print("puesto ingr.: ",i);  
  end  
end
```

```
process HebraFumador[ i : 0..2 ]  
  var b : integer ;  
begin  
  while true do begin  
    { Sentencia  $R_i$  : }  
    write("retirado ingr.:",i);  
    { simular el fumar: }  
    Fumar( i );  
  end  
end
```

- ▶ El estancuero produce un número aleatorio mediante una llamada a la función **Producir**, que conlleva un cierto retraso aleatorio y devuelve un entero (0,1 o 2).
- ▶ Los fumadores fuman llamando a la función **Fumar**, que conlleva un retraso aleatorio.

*Hacen falta 4 senaforas*

# Condición de sincronización y semáforos (1)

$(P_0, P_1, P_2, R_0, R_1, R_2)$

Hay un total de 6 instrucciones que se ejecutan repetidamente ( $P_i$  y  $R_i$ , tres en cada caso). Pero no podemos permitir cualquier interfoliación de las mismas. En concreto:

- ▶ Para cada  $i$ , el número de veces que ha el fumador número  $i$  ha retirado el ingrediente número  $i$  no puede ser mayor que el número de veces que se ha producido el ingrediente  $i$ , es decir  $\#R_i \leq \#P_i$ , o lo que es lo mismo:

$$S_0 = \#P_0 - \#R_0$$

$$S_1 = \#P_1 - \#R_1$$

$$S_2 = \#P_2 - \#R_2$$

$$0 \leq \#P_i - \#R_i$$

← Son 3 condiciones  
(una para cada ingrediente)

- ▶ Esto se puede resolver con tres semáforos  $s_i$  (un array de semáforos, con  $i = 0, 1, 2$ ), cada uno de ellos vale  $\#P_i - \#R_i$ .
- ▶ En el semáforo  $s_i$  se debe de hacer:
  - ▶ **sem\_wait** antes de  $R_i$  (aparece con signo negativo)
  - ▶ **sem\_signal** después de  $P_i$  (aparece con signo positivo)

Cada semáforo indica si el ingr. n.º  $i$  está (o no) disponible en el mostrador.

## Condición de sincronización y semáforos (2)

La condición anterior no contempla todas las restricciones del problema, hay que añadir esta: *(Otro semáforo más)*

- ▶ El número  $p$  de ingredientes producidos (en el mostrador) y pendientes de ser usados por algún fumador es

$$p = \sum_{i=0}^2 \#P_i - \sum_{i=0}^2 \#R_i$$

- ▶ Solo cabe un ingrediente como mucho en el mostrador, luego  $p$  solo puede ser 0 o 1. Es decir, se debe cumplir (a)  $0 \leq p$  y (b)  $p \leq 1$ . *Solo cabe 1 ingrediente* *Se cumple con las sem. anteriores.*
- ▶ La condición (a) esta garantizada por la condición de la transparencia anterior, así que solo hay que asegurar (b), que se puede escribir como:

$$0 \leq 1 + \sum_{i=0}^2 \#R_i - \sum_{i=0}^2 \#P_i$$

## Condición de sincronización y semáforos (3)

Por tanto, podemos usar un semáforo (lo llamamos **mostr\_vacio**), cuyo valor es

$$1 + \sum_{i=0}^2 \#R_i - \sum_{i=0}^2 \#P_i$$

Es decir, vale 1 si el mostrador está libre, y 0 si hay un ingrediente en él. Sobre ese semáforo, debemos de hacer:

- ▶ **sem\_wait** antes de cualquier sentencia  $P_i$  (ya que aparecen con signo negativo)
- ▶ **sem\_signal** después de cualquier sentencia  $R_i$  (ya que aparecen con signo positivo)

Las sentencias  $P_i$  y  $R_i$  no suponen asignación ninguna (no hacen nada).

# Esquema de las hebras

Por tanto, el esquema de las hebras, incluyendo la sincronización, será este:

**tenemos un array de hebras y otro de semáforos**  
**hay un semáforo para cada fumador (en un array de semáforos)**

```
{ variables compartidas y valores iniciales }  
var mostr_vacio : semaphore := 1 ; { 1 si mostrador vacío, 0 si ocupado }  
    ingr_disp   : array[0..2] of semaphore := { 0,0,0 } ;  
                { 1 si el ingrediente i esta disponible en el mostrador, 0 si no }
```

```
process HebraEstanquero ;  
  var i : integer ;  
begin  
  while true do begin  
    i := Producir();  
    sem_wait( mostr_vacio );  
    print("puesto ingr.: ",i); {Pi}  
    sem_signal( ingr_disp[i] );  
  end  
end
```

Antes de poner el ingrediente al estancuero  
debe esperar a que el mostrador este libre

```
process HebraFumador[ i : 0..2 ]  
begin  
  while true do begin  
    sem_wait( ingr_disp[i] );  
    print("retirado ingr.:",i); {Ri}  
    sem_signal( mostr_vacio );  
    Fumar( i );  
  end  
end
```

No se puede retirar un ingrediente hasta  
que este puesto en el mostrador por el estancuero

```
{ variables compartidas y valores iniciales }
var mostr_vacio : semaphore := 1 ; { 1 si mostrador vacio, 0 si ocupado }
    ingr_disp : array[0..2] of semaphore := { 0,0,0 } ;
    { 1 si el ingrediente i esta disponible en el mostrador, 0 si no }
```

```
process HebraEstanquero ;
var i : integer ;
begin
    while true do begin
        i := Producir();
        sem_wait( mostr_vacio );
        print("puesto ingr.: ", i); {Pi}
        sem_signal( ingr_disp[i] );
    end
end
```

Antes de poner al ingrediente al estancoero  
debes esperar a que el mostrador este libre

```
process HebraFumador[ i : 0..2 ]
begin
    while true do begin
        sem_wait( ingr_disp[i] );
        print("retirado ingr.: ", i); {Ri}
        sem_signal( mostr_vacio );
        Fumar( i );
    end
end
```

No se puede retirar un ingrediente hasta  
que este puesto en el mostrador por el estancoero

## HEBRA ESTANQUERO

while (true)

num\_fumador = producir\_ingred();

cout << "Est: empieza prod..."  
cout << "Est: termina prod ingr {0,1,2},  
random

SEM-WAIT( mostr\_vacio);

cout << "Puesto ingrediente {0,1,2}";  
num\_fumador

SEM-SIGNAL( ingr\_disp[num\_fumador]);

## (hay varias)

## HEBRA FUMADOR

(porau → num\_fumador)

while (true)

SEM-WAIT( ingr\_disp[num\_fumador]);

cout << "Retirado ingrediente {0,1,2}";  
num\_fumador

SEM-SIGNAL( mostr\_vacio);

fumar( num\_fumador);

cout << "Fumador {0,1,2}: empieza a fumar";  
cout << "Fumador {0,1,2}: termina, comienza espera de ingrediente";  
num\_fumador

Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 4. El problema de los fumadores.

Subsección 4.2.

**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, y produce un retraso aleatorio.

```
// función que simula la acción de fumar, como un retardo aleatorio de la hebra.  
// recibe como parámetro el numero de fumador  
void fumar( int num_fum )  
{  
    cout << "Fumador número " << num_fum << ": comienza a fumar." << endl;  
    this_thread::sleep_for( chrono::milliseconds( aleatorio<50,200>() ) );  
    cout << "Fumador número " << num_fum << ": termina de fumar." << endl;  
}  
  
// funciones que ejecutan las hebras  
void funcion_hebra_estanquero( ) { .... }  
void funcion_hebra_fumador( int num_fum ) { .... }  
  
int main()  
{  
    // poner en marcha las hebras y esperar que terminen .....  
}
```



Sistemas Concurrentes y Distribuidos., curso 2022-23.

**Práctica 1. Sincronización de hebras con semáforos.**

Sección 4. El problema de los fumadores.

Subsección 4.3.

**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.
  - ▶ 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. Nombres de los semáforos empleados para sincronización y, para cada uno de ellos:
  - ▶ Utilidad.
  - ▶ Valor inicial.
  - ▶ Hebras que hacen **wait** y **signal** sobre dicho semáforo.
2. Código fuente completo de la solución adoptada.

Fin de la presentación.