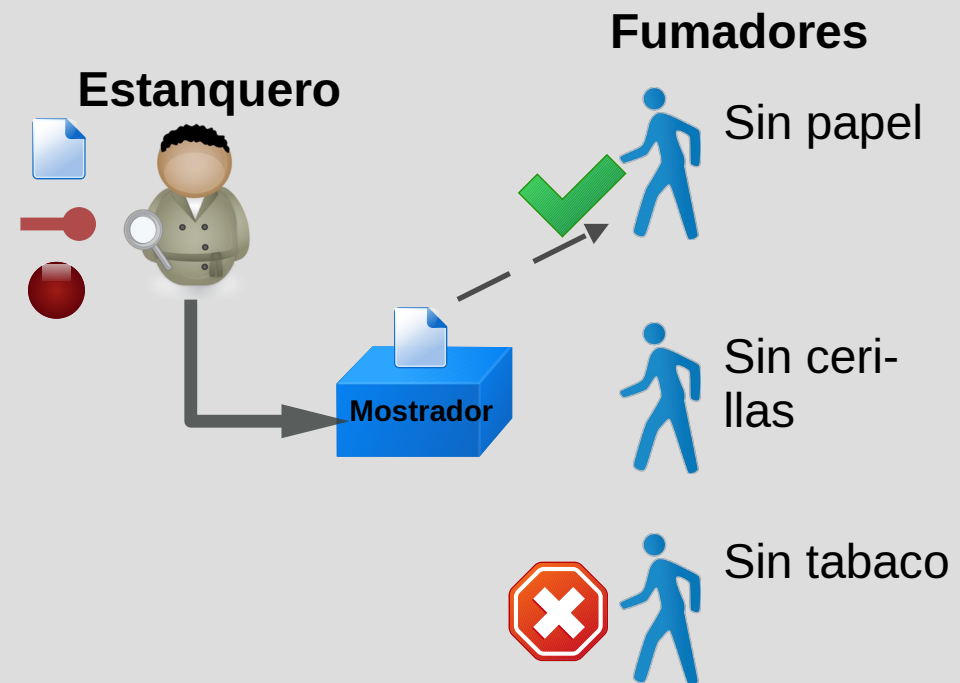
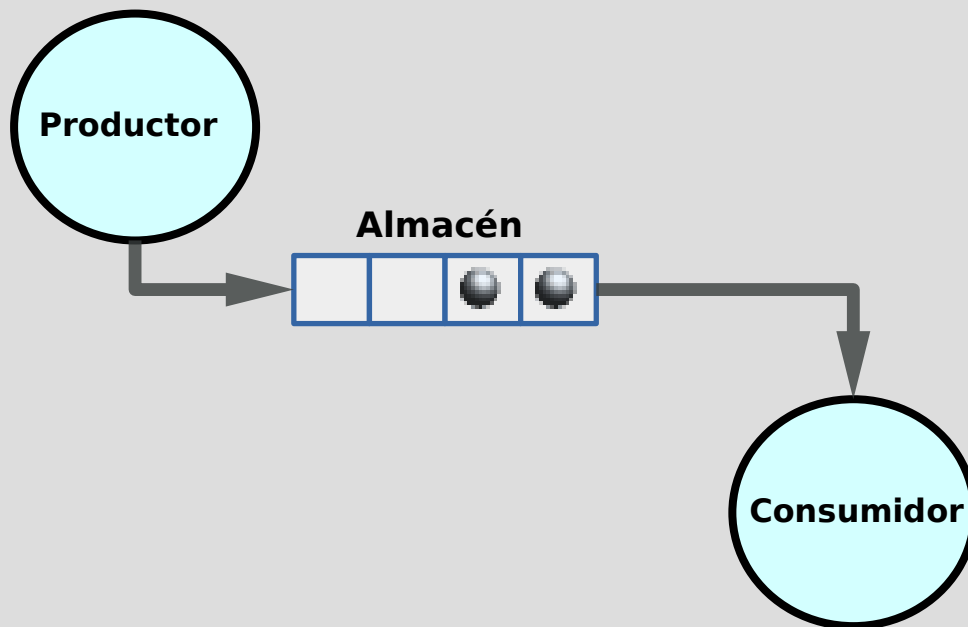


Práctica 1: Sincronización de hebras con semáforos

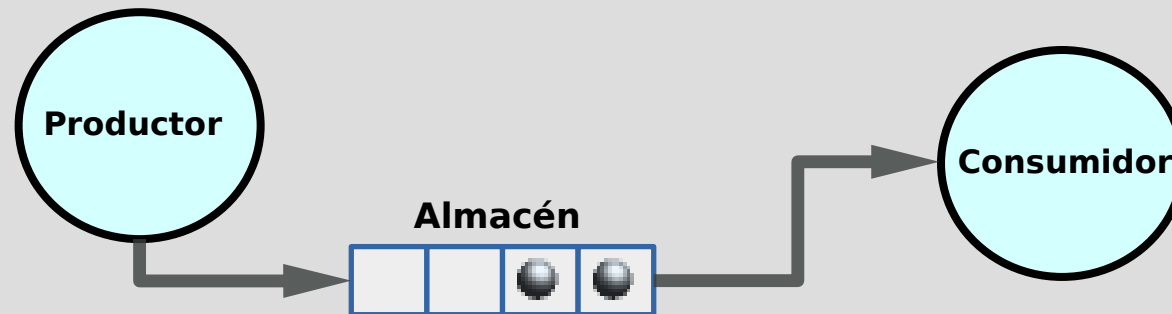
1. Objetivos
2. Espera bloqueada de una hebra
3. El problema del productor-consumidor
4. El problema de los múltiples productores y consumidores
5. El problema de los fumadores.



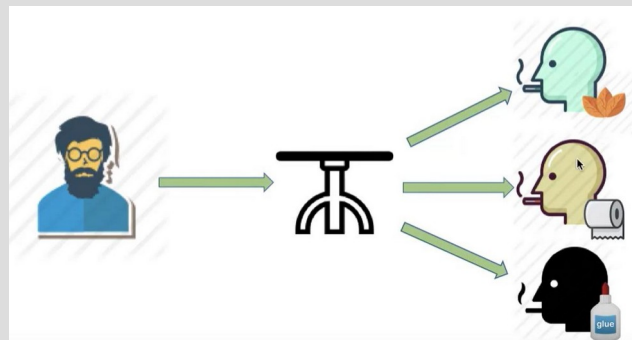
1. Objetivos

Se abordará la **implementación concurrente** de dos problemas sencillos de sincronización:

a) Problema del Productor-Consumidor (con relaciones 1-1 y muchos-muchos)



b) Problema de los fumadores (relación 1-muchos con destino fijado)



Se usarán librerías abiertas para programación multihebra y para usar semáforos como medio de sincronización.

2. Espera bloqueada de una hebra

Generación de números aleatorios: usaremos la **plantilla de función “aleatorio”**, disponible en **scd.h**

```
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
```

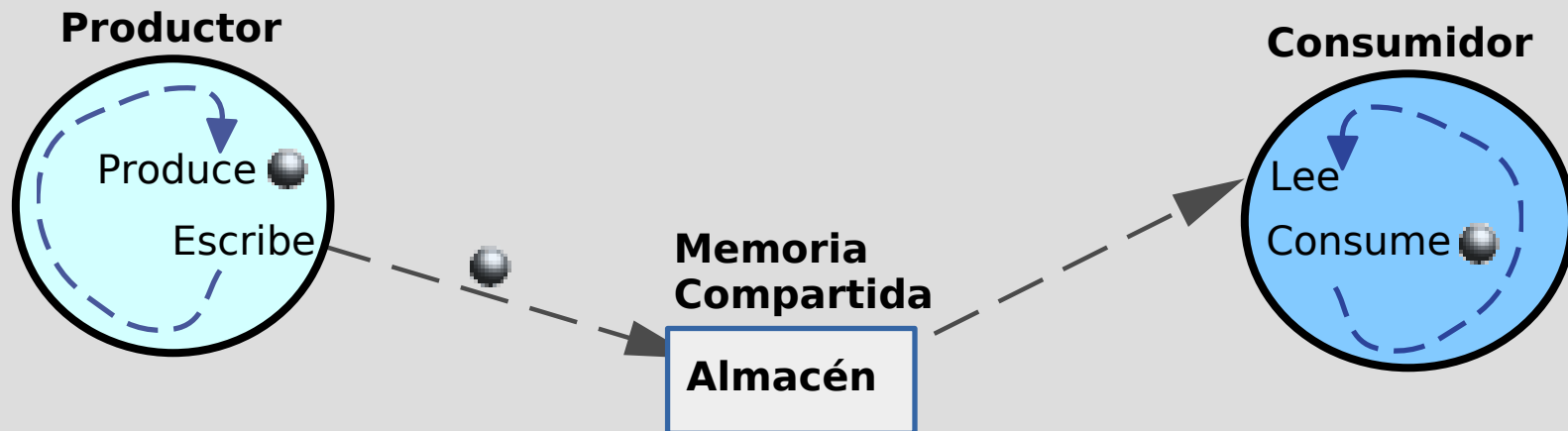
Esperas de duración aleatoria en las hebras

- Permiten **simular la realización de trabajo útil** por parte de las hebras durante cierto tiempo.
- Se usará el método **sleep_for** de la clase **this_thread** cuyo argumento es un valor de tipo **duration**.

```
// 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 );
```

3. El problema del Productor-Consumidor Descripción (1)

Un proceso/hebra produce items de datos en memoria y otro proceso/hebra los consume



Ejemplo: Aplicación de reproducción de vídeo

- **Productor** lee de disco y descodifica cada cuadro de vídeo.
- **Consumidor** lee cuadros descodificados y los envía a mem. de vídeo para visualización.

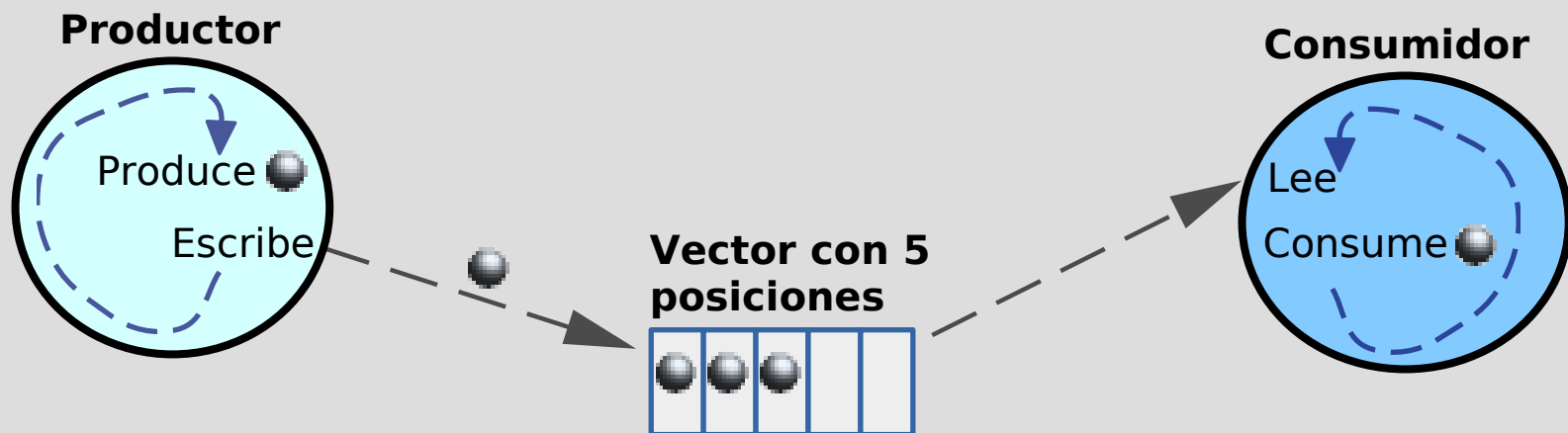
El tiempo que se tarda en producir/consumir un item de datos puede ser variable.

3. El problema del Productor-Consumidor

Descripción (2)

El desarrollo de un programa concurrente para este problema:

- Productor y consumidor se implementan como **dos hebras independientes**, permitiendo el aprovechamiento de dos CPUs disponibles.
- Se puede usar una **única variable compartida** con capacidad para un ítem de datos, pero las esperas asociadas pueden **empeorar la eficiencia**.
- **Mejora:** usar un **vector que pueda contener varios ítems** de datos producidos y pendientes de leer.



3. El problema del Productor-Consumidor

Esquema de dos hebras sin sincronización

```
{ 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
```

3. El problema del Productor-Consumidor.

Condición de sincronización

El programa concurrente debe garantizar que;

Cada ítem producido es leído y una única vez.

- **Productor espera** antes de poder escribir un nuevo ítem producido si vector está lleno.

- **Consumidor espera** antes de leer un ítem si vector vacío.

- En algunas aplicaciones: el orden de lectura podría tener que coincidir con el de escritura.

- No se debe impedir que **ProducirValor()** y **ConsumirValor()** se solapen en el tiempo.

```
{ 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
```

3. El problema del Productor-Consumidor

Solución con semáforos

```
{ 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 );
    { Sentencia E:                }
    { (insertar valor 'a' en 'vec') }
    sem_signal( ocupadas );
  end
end
```

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

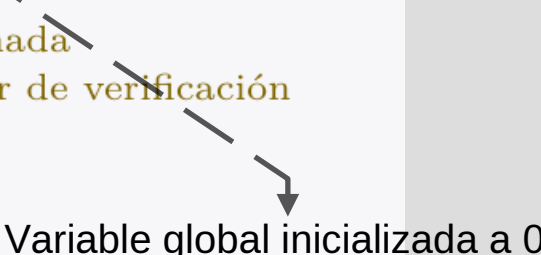

3. El problema del Productor-Consumidor

Trabajo Propuesto. Producir y consumir datos

Programa concurrente en C/C++:

- Productor produce valores enteros (int) en secuencia, empezando en 1.
- El orden en el que consumidor lee los items es irrelevante (en principio).
- Consumidor escribe cada valor leído en pantalla.
- Se usará un vector compartido de enteros (int), de tamaño fijo.
- Hebra productora llama a **producir_dato** y hebra consumidora a **consumir_dato**:

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




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

3. El problema del Productor-Consumidor

Trabajo propuesto. Funciones para las hebras

```
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 ) ;
    }
}
```

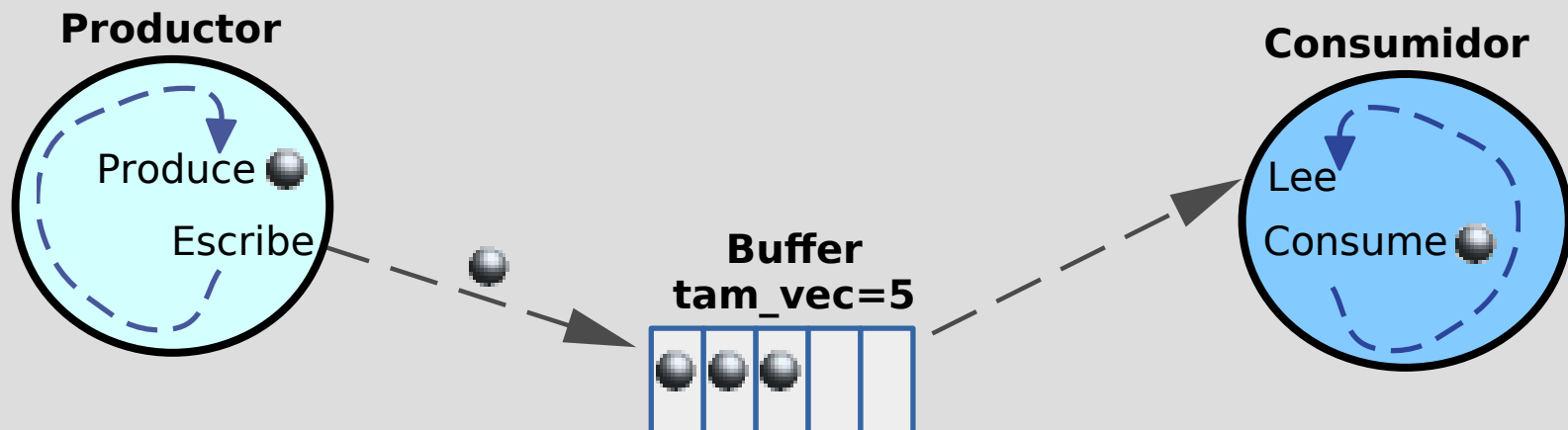


Inicializar a 50, 100, ...

3. El problema del Productor-Consumidor

Gestión ocupación vector intermedio (1)

- El **vector intermedio** (buffer) será un array C++ de tamaño fijo igual a la constante **tam_vec**.
- En cualquier instante, el **número de celdas ocupadas** en el vector es un número entre 0 (buffer vacío) y **tam_vec** (buffer lleno).
- Además del vector, debemos usar algunas **variables adicionales** que reflejen el estado de ocupación.
- El **acceso a dichas variables** puede requerir sincronización entre productor y consumidor.

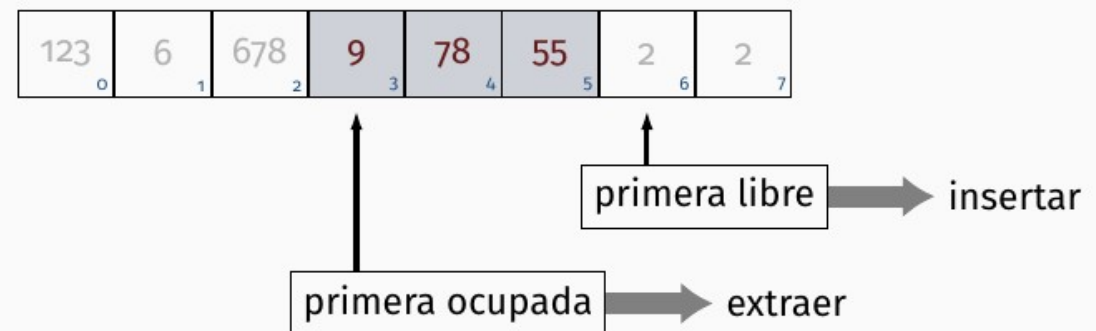
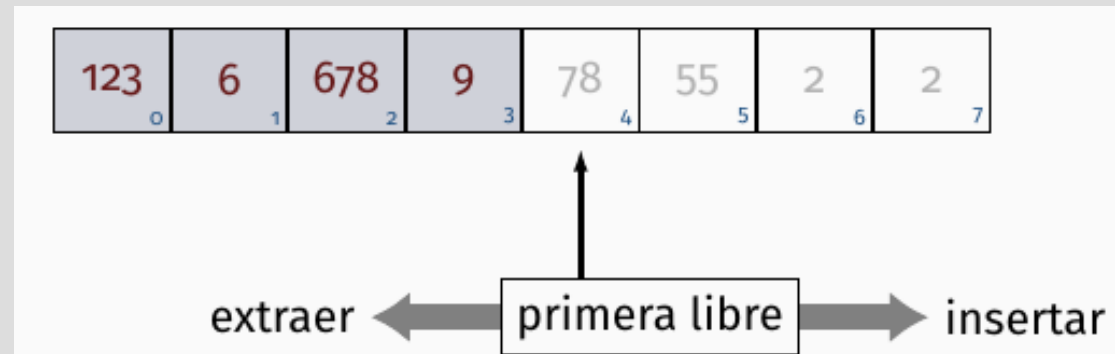


3. El problema del Productor-Consumidor

Gestión ocupación vector intermedio (2)

Hay **dos alternativas** para gestionar la ocupación del buffer:

- **LIFO** (pila acotada): requiere usar una variable entera:
 - **primera_libre** = índice de la primera celda libre
- **FIFO** (cola circular): Usa dos variables enteras:
 - **primera_ocupada** = índice de la primera celda ocupada (inicialmente 0). Se incrementa al leer (módulo tam_vec).
 - **primera_libre** = índice primera celda libre (inicialmente 0). Se incrementa al escribir (módulo tam_vec).



3. El problema del Productor-Consumidor

Actividades

Implementa la solución descrita en un programa C++ con hebras C++11 y usando la biblioteca de semáforos, completando las plantillas incluidas.

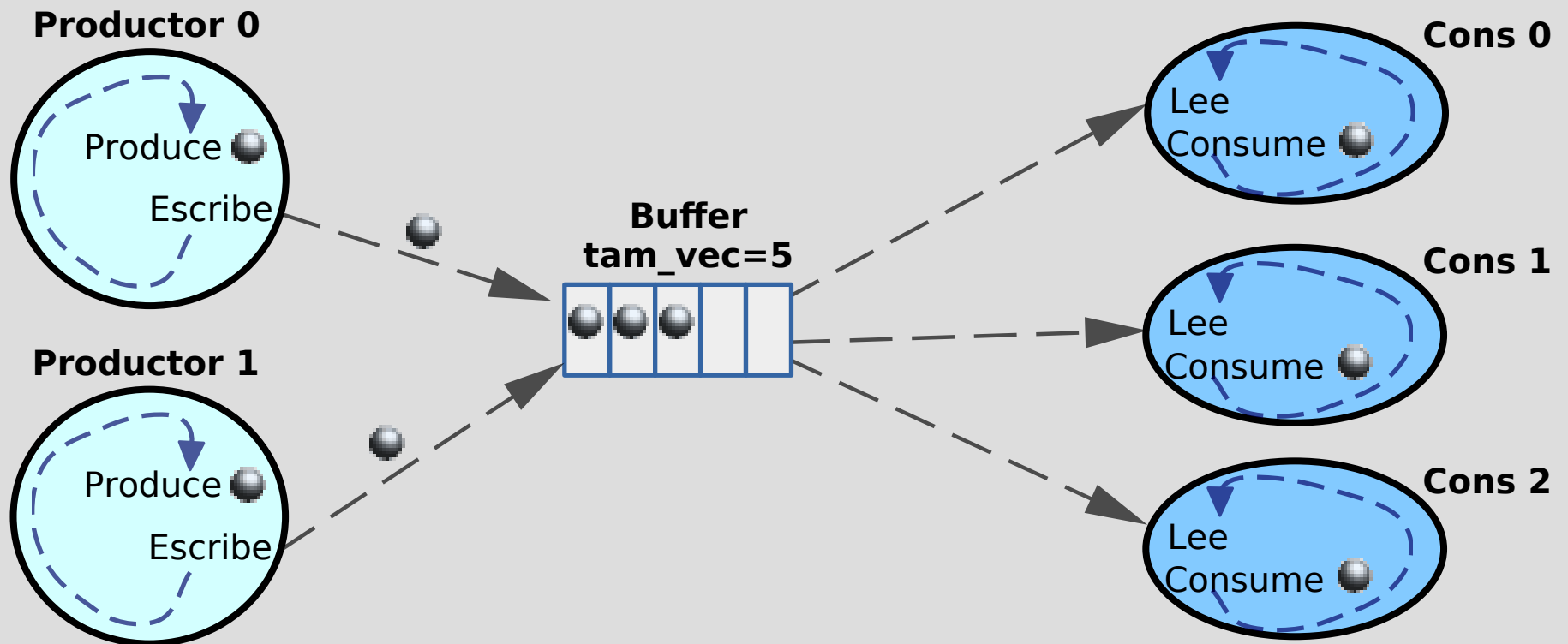
- **Comprueba que tu programa es correcto:** verifica que cada número natural producido es consumido exactamente una vez.
- Es conveniente **incluir sentencias para imprimir en pantalla** el valor insertado/extraído del buffer, justo después de cada inserción/extracción.
- **CUIDADO!!:** Los mensajes en pantalla no tienen porqué reflejar el estado cuando aparecen, sino un estado antiguo distinto del actual.
- El programa debe escribir la **palabra fin** cuando hayan terminado las dos hebras.

4. El problema de los múltiples Productores y Consumidores

Descripción

Extensión de la solución descrita:

- Permitir más de una hebra productora y más de una hebra consumidora.
- Varios productores/consumidores pueden estar produciendo/consumiendo simultáneamente.
- La **gestión del vector compartido** es prácticamente similar a la anterior: **Opciones LIFO/FIFO**

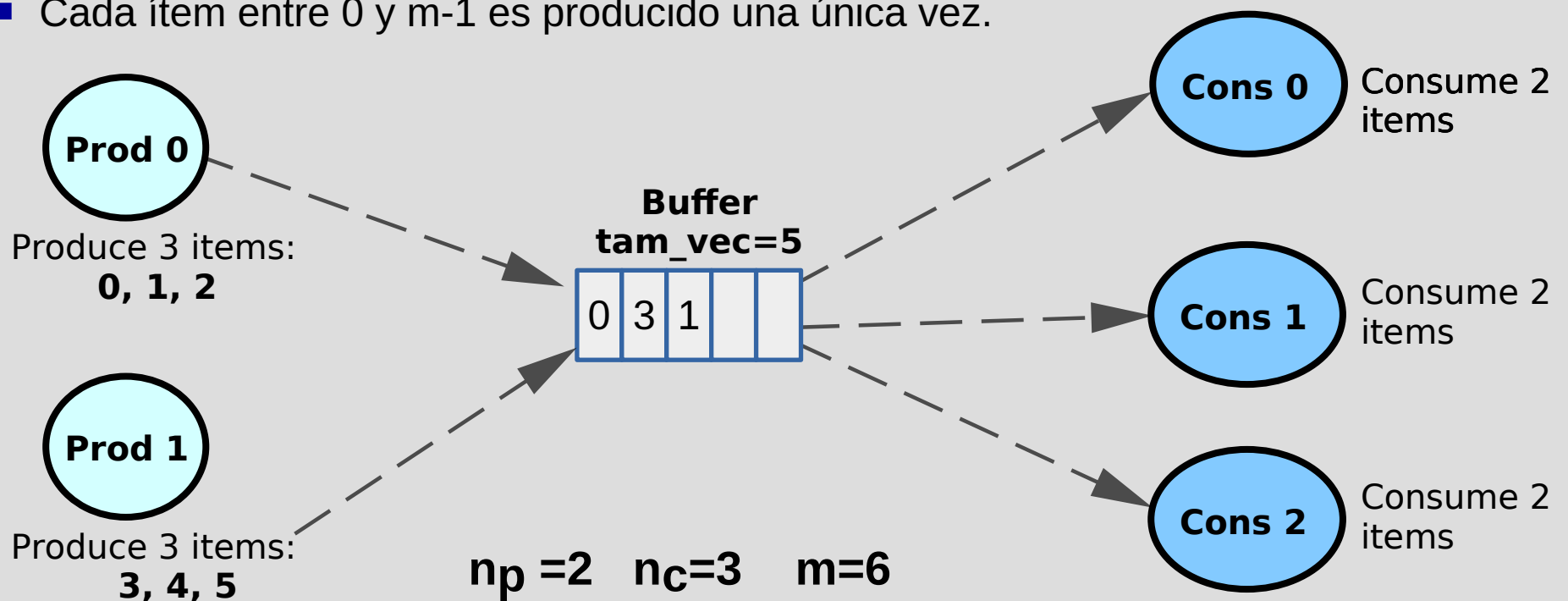


4. El problema de los múltiples Productores y Consumidores

Actividades

Copia archivo **prodcons.cpp** a **prodcons-multi.cpp** e implementa en el nuevo archivo la extensión:

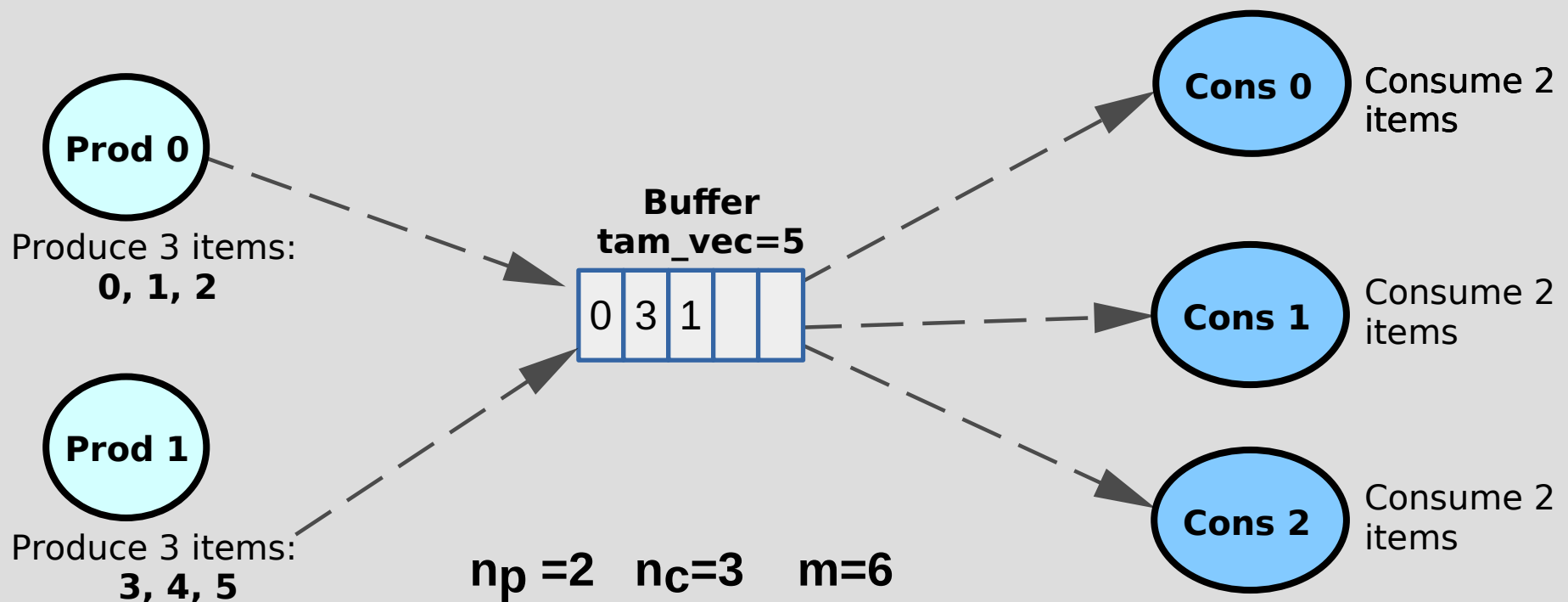
- Habrá n_p hebras productoras y n_c hebras consumidoras (constantes del programa).
- El número total de items producidos/consumidos será igual a m , múltiplo de n_p y n_c .
- Cada productor produce $p=m/n_p$ items y Cada consumidor consume $c=m/n_c$ items.
- Cada ítem entre 0 y $m-1$ es producido una única vez.



4. El problema de los múltiples Productores y Consumidores

Requisitos

- Funciones **producir_dato** y **consumir_dato** tienen como argumento el número de hebra productora ($i=0, \dots, n_p - 1$) o consumidora ($j=0, \dots, n_c - 1$).
- La i -ésima hebra productora produce los items desde $i \cdot p$ hasta $i \cdot p + (p-1)$.
- Se puede declarar un array global compartido con n_p celdas, que se lee y escribe en **producir_dato**, que indica cuántos items se han producido ya. Debe inicializarse a 0 y la i -ésima hebra solo accede a la celda i -ésima del mismo.



4. El problema de los múltiples Productores y Consumidores

Resolución de conflictos de acceso: Exclusión mutua

Se deben **resolver los conflictos de acceso** al buffer compartido **asegurando la EM** pero sin perder concurrencia:

- 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.

- En la **solución FIFO**:

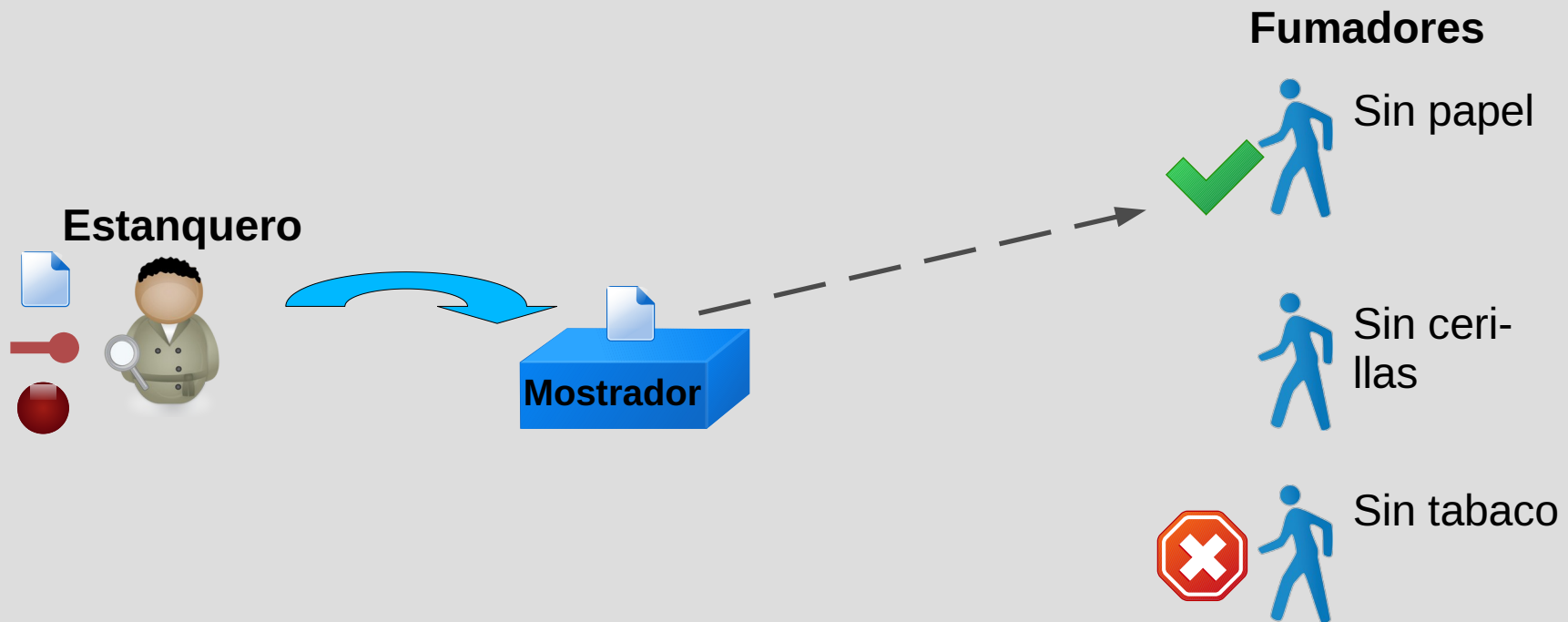
- Dos o más hebras productoras pueden intentar leer y modificar a la vez la variable **primera_libre** y la correspondiente celda.
- Dos o más hebras consumidoras pueden intentar leer y modificar a la vez la variable **primera_ocupada** y la correspondiente celda.

5. El problema de los Fumadores

Descripción del problema (1)

Estanco con 3 fumadores y un estancquero (hebras)

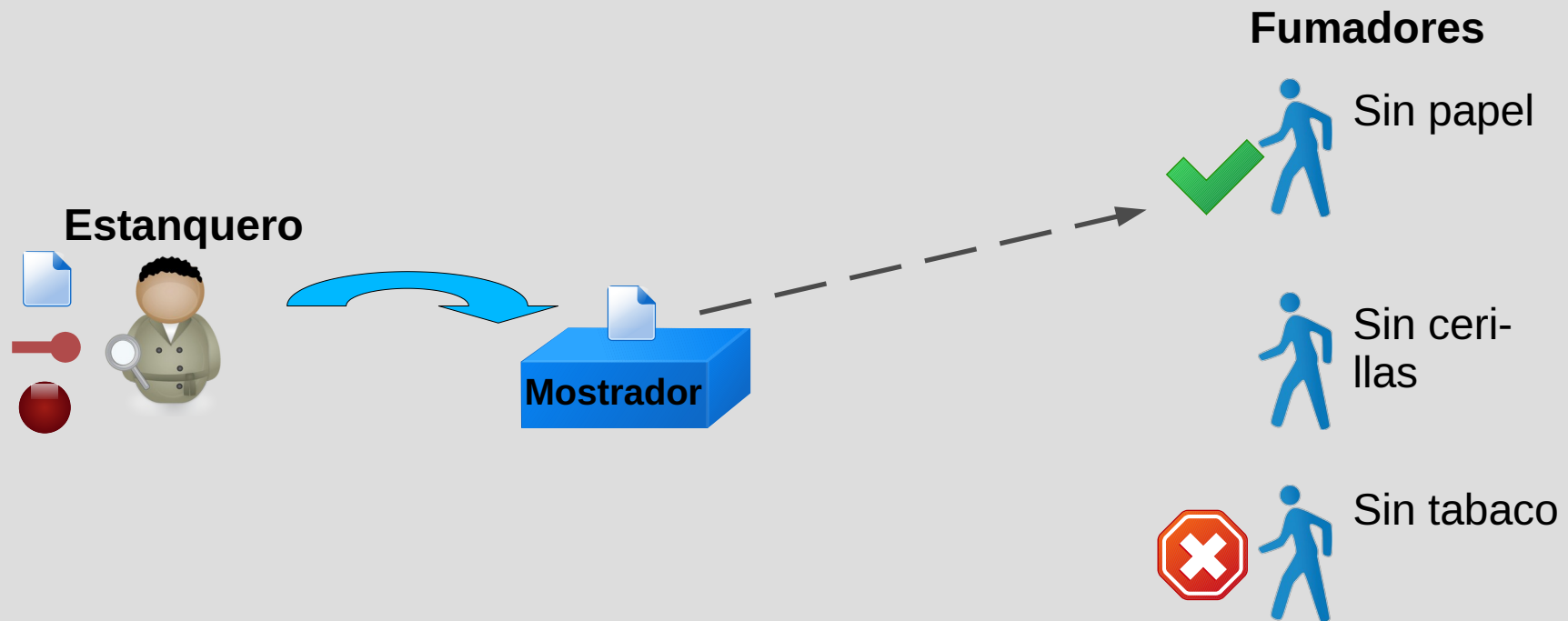
- Los **fumadores** están en un **bucle infinito** donde invocan la función **fumar**
- Cada fumador debe tener suministros antes de fumar.
- El **estancquero produce un ítem** de suministro cada vez (papel, cerillas o tabaco) en bucle infinito.
- La solución debe **permitir que varios fumadores fumen simultáneamente**.



5. El problema de los Fumadores

Descripción del problema (2)

- Cada **fumador necesita 3 ingredientes** para fumar (papel, cerillas y tabaco) pero necesita uno para fumar y cada fumador necesita uno distinto.
- En cada iteración, **Estanquero selecciona uno aleatoriamente**, lo pone en mostrador avisando al fumador correspondiente y esperando a que lo retire.
- Cuando un **fumador detecta** su ingrediente, lo coge **avisando** al estancuero y fuma un tiempo aleatorio.
- Una vez avisado, el estancuero pasa al siguiente.



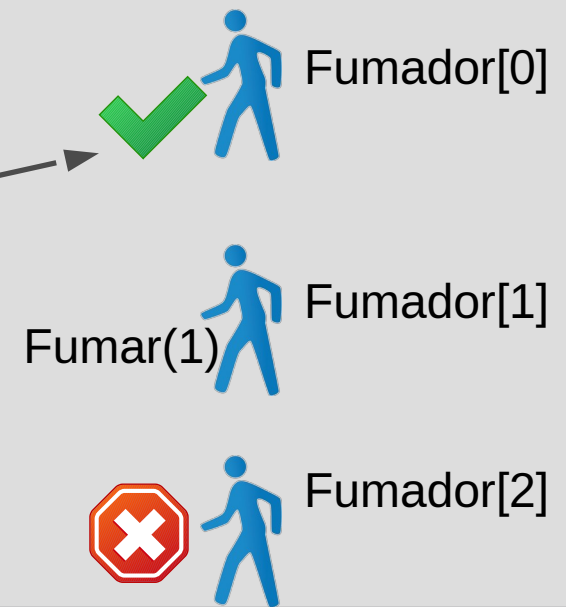
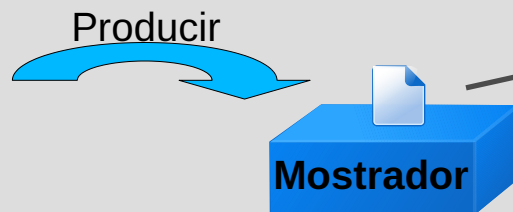
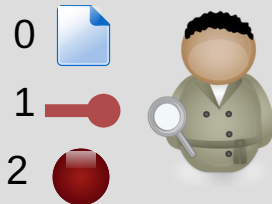
5. El problema de los Fumadores

Esquema de las hebras

```
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
```

Ingredientes



5. El problema de los Fumadores

Sincronización

```
{ 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
```

```
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
```

5. El problema de los Fumadores

Función Fumar(). Simulación de la acción de fumar

```
// 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 .....  
}
```

5. El problema de los Fumadores

Actividades

Diseña e implementa una **solución al problema en C++** usando cuatro hebras y los semáforos necesarios. La solución debe cumplir los requisitos descritos y:

- **Evitar interbloqueos** entre las distintas hebras.
- **Producir mensajes en la salida estándar** que permitan seguimiento:
 - El estancero debe indicar cuándo produce y qué ítem produce.
 - Cada fumador debe indicar cuándo espera, qué ítem espera, y cuándo comienza y finaliza de fumar.