UD 3.- Primitivas de sincronización

Concurrencia y Sistemas Distribuidos

Objetivos de la Unidad Didáctica

- Explicar el modelo de programación concurrente proporcionado por los monitores.
- Solucionar el problema de la sincronización condicional mediante la utilización de monitores.
- Construir monitores en lenguajes de programación concurrente, evitando sus problemas potenciales.
- Evaluar las variantes de monitor existentes.

Contenido

- Lenguajes de programación concurrente
- Concepto de monitor
- Variantes de monitor
- Invocaciones cruzadas

Lenguajes de Programación Concurrente

- La Programación Concurrente debe resolver las necesidades de comunicación y sincronización entre hilos
- Podemos diseñar estrategias de solución a varios niveles:

Sin soporte del Sistema Operativo

- Espera activa
- Inhabilitación de interrupciones (en modo supervisor)
- Primitivas *TestAndSet*, Swap

Con soporte del Sistema Operativo

- Semáforos

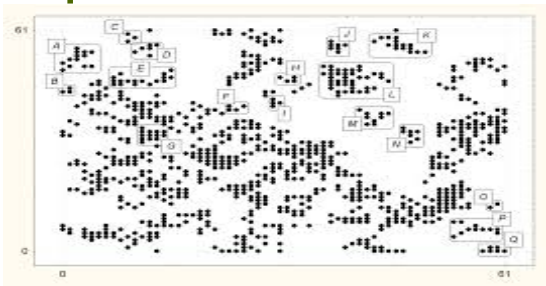
➢ Pros: Eficiente y flexible
➢ Cons: Su uso incorrecto no es detectado en programación

Utilizando Lenguajes de Programación Concurrente

- Monitores

Ejemplo.- Hormigas

- Las hormigas comparten un territorio
  - Matriz de celdas, cada una con valores libre/ocupado
  - Máximo una hormiga por celda
  - Existe un lock que protege este territorio



- Cada hormiga se modela mediante un hilo

- Cuando una hormiga se desplaza desde (x,y) a (x',y') ejecuta

```
cerrar lock
// Si ocupada[x',y'] debe esperar hasta que quede libre
ocupada[x',y']=true; ocupada[x,y]=false //actualiza matriz
abrir lock
```

¿Cómo realizamos esta espera?

Monitor.- motivación

- Las primitivas ***abrir lock*** y ***cerrar lock*** garantizan el acceso seguro a variables compartidas
- Pero necesitamos **otras primitivas** que permitan **esperar** de forma segura **hasta que se cumpla** determinada **condición lógica** (sincronización)

  - Un bucle de espera (bucle vacío que comprueba repetidamente la condición) no funciona. ¿Por qué?

```
cerrar lock;
while (ocupada[x',y']) {} //NO funciona
ocupada[x',y']=true; ocupada[x,y]=false;
abrir lock;
```

Monitor.- motivación

- La mayor parte de los Lenguajes de Programación modernos:
  - Son lenguajes orientados a objetos
    - El programador puede definir tipos de datos (clases)
    - Una clase permite definir variables (objetos)
    - Separación interfaz/implementación
      - La interfaz (parte visible) corresponde a su comportamiento (conjunto de métodos)
      - Implementación (parte oculta).- conjunto de atributos, código de los métodos
  - Requieren concurrencia

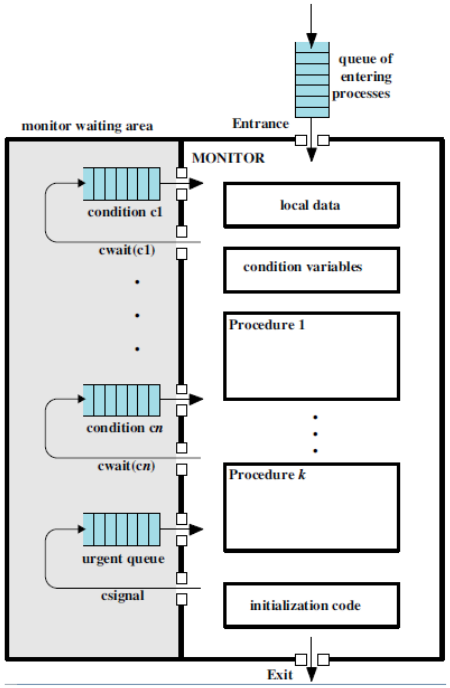
Monitor.- motivación

- Idea.- mezclar POO y Programación Concurrente
  - Los hilos se coordinan mediante **objetos compartidos**
  - Ocultamos los detalles de exclusión mutua y sincronización en las clases que representan los objetos compartidos

- Ventajas
  - Simplifica el desarrollo, mantenimiento, y comprensión del código
  - Facilita la depuración (podemos probar cada pieza por separado)
  - Facilita la reutilización de código
  - Mejora la documentación y la legibilidad

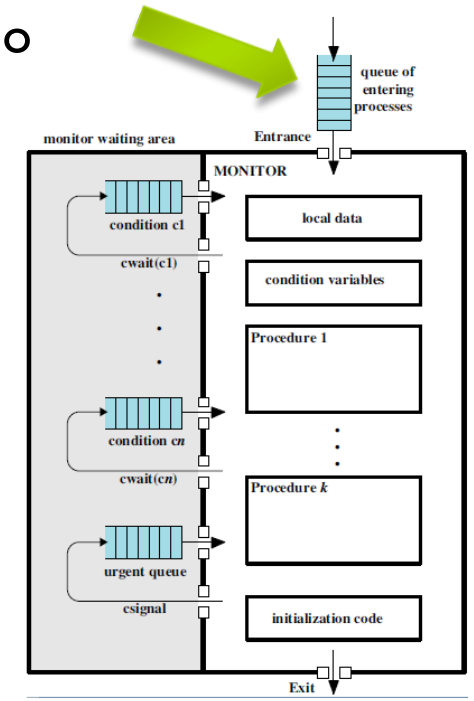
Monitor

- **Monitor** = clase para definir objetos que podemos compartir de forma segura entre distintos hilos
  - Sus métodos se ejecutan en exclusión mutua
  - Resuelve la sincronización



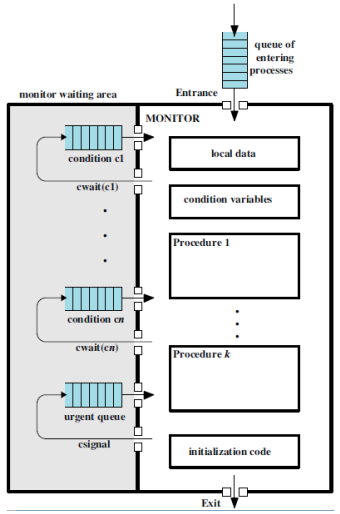
Monitor

- **Monitor** = clase para definir objetos que podemos compartir de forma segura entre distintos hilos
  - Sus métodos se ejecutan en exclusión mutua
    - Dispone de una **cola de entrada** donde esperan aquellos hilos que desean utilizar el monitor cuando lo está utilizando otro hilo
    - **No hay condiciones de carrera dentro del monitor**

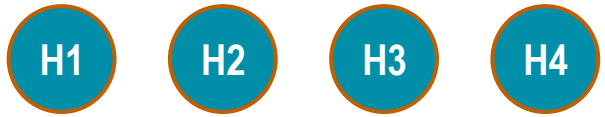


Monitor

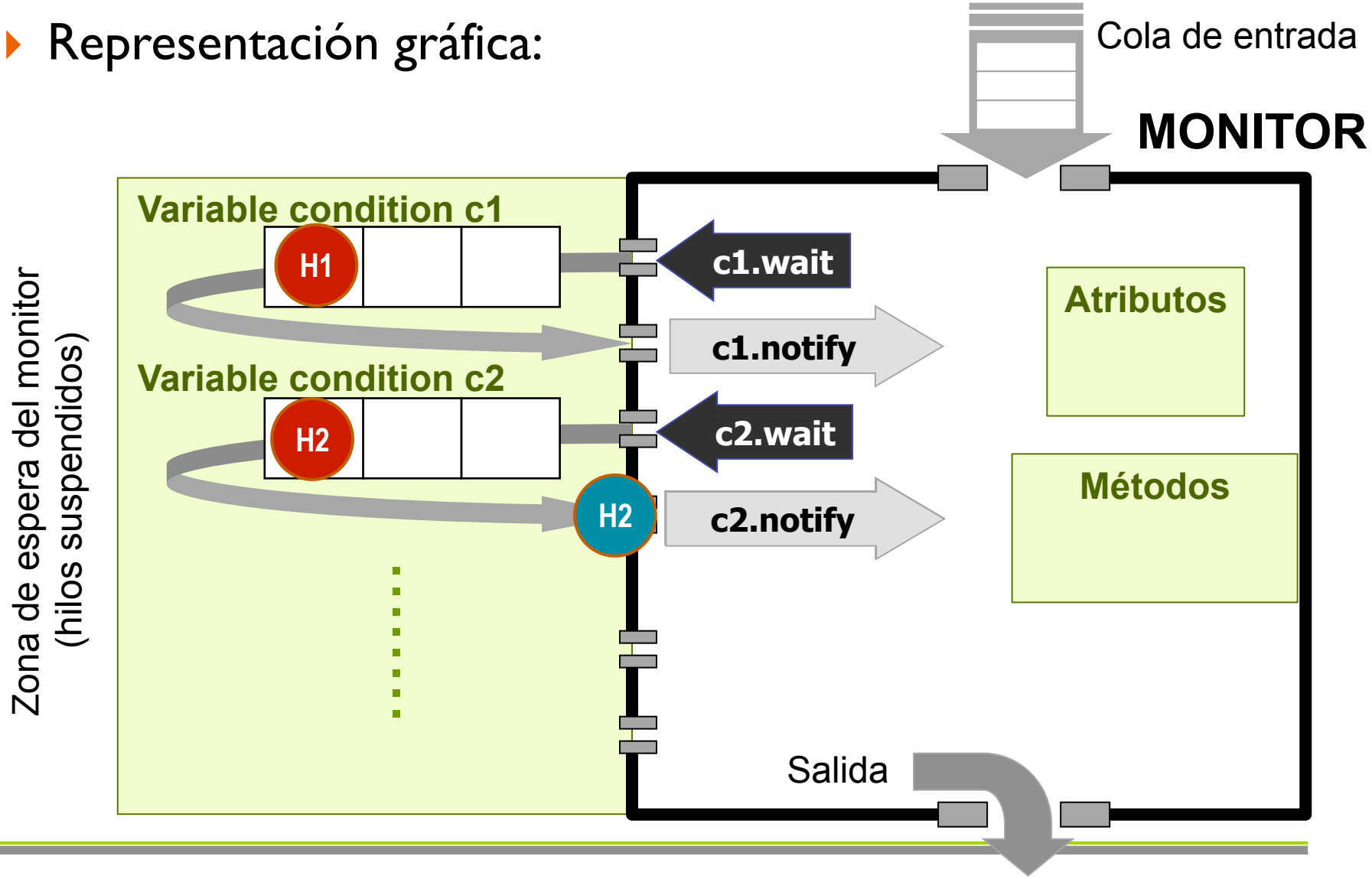
- **Monitor** = clase para definir objetos que podemos compartir de forma segura entre distintos hilos
  - Resuelve la sincronización
    - Podemos definir **colas de espera** (variables '**condition**') dentro del monitor
    - Si un hilo que ejecuta código dentro del monitor necesita **esperar** hasta que se cumpla determinada condición lógica
      - Ejecuta **c.wait()** → deja libre el monitor y espera sobre la cola de la condición *c*
    - Cuando otro hilo **modifica el estado** del monitor
      - Ejecuta **c.notify()** → reactiva un hilo que espera en la cola de la condición *c*



Concepto de Monitor



- Representación gráfica:



Ejemplo Monitor.- Productor/Consumidor

- El objeto compartido es el **buffer**
  - Diseñamos interfaz e implementación (atributos y código de los métodos) como en cualquier otra clase
- Rellenamos una tabla en la que detallamos:

Método	Espera cuando	Avisa a
int get()	Buffer vacío	Quien espera por buffer lleno
void put(int e)	Buffer lleno	Quien espera por buffer vacío
int numElems()	--	--

- Definimos una cola de espera (**variable condition**) por cada caso de espera en la tabla
  - condition noLleno, noVacio;

Ejemplo Monitor.- Productor/Consumidor

```
Monitor  Buffer {                    //IMPORTANTE.- NO ES JAVA, sino pseudolenguaje
   .... //atributos para implementar el monitor
  condition noLleno, noVacio;  //colas de espera
  int elems = 0;

  public Buffer() {..} //inicialización de los atributos
  entry void put(int x) { // entry= método público con acceso en exclusión mutua
     if (elems==N) {noLleno.wait();}    // espera en la cola noLleno
        ...  //código para insertar
     elems++;
     noVacio.notify();                  // reactiva a alguien de la cola noVacio
  }
  entry int get() {
     if (elems==0) {noVacio.wait();}    // espera en la cola noVacio
        ... // código para extraer
     elems--;
     noLleno.notify(); return ... ;     // reactiva a alguien de la cola noLleno
  }
  entry int numItems() {return elems;}
}
...
Buffer b; //y desde cualquier hilo se puede invocar b.numItems(), b.get() o b.put(x)
```

Monitor.- resumen

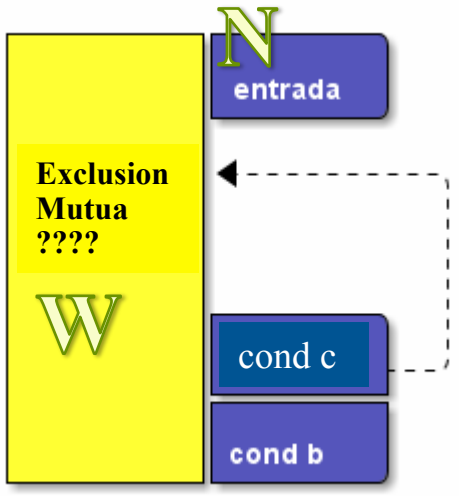
- Monitor = **Clase + Exclusión Mutua + Sincronización**
- Oculta los detalles de Exclusión Mutua y sincronización
  - Ejecución de métodos en el mismo monitor en exclusión mutua
  - Para la coordinación se usan colas de espera (variables *condition*)
    - Primitivas: *wait()* para esperar; *notify()* para notificar a quien espera
      - Si al avisar no hay nadie esperando, el aviso se pierde (no tiene efecto)
    - Únicamente pueden utilizarse dentro del monitor
    - El programador es responsable de esperar/avisar en los momentos oportunos
- Proporciona abstracción
  - El programador que invoca operaciones sobre el monitor ignora cómo se implementan
  - El programador que implementa el monitor ignora cómo se usa

Contenido

- Lenguajes de programación concurrente
- Concepto de monitor
- Variantes de monitor
- Invocaciones cruzadas

Monitor.- Variantes

- El monitor garantiza la **Exclusión Mutua**
  - Sólo un hilo ejecuta código del monitor en un instante dado
  - Cuando el **hilo activo** (W) en el monitor ejecuta **c.wait()**, pasa a esperar sobre "c"
    - **El monitor queda libre** (espera fuera de la SC)
    - Otro hilo (N) que espera en la entrada pasa a activo en el monitor
- Problema: si el hilo N ejecuta c.*notify()*
  - Reactiva a W
  - Pero sólo uno (W o N) puede continuar activo en el monitor. ¿Cuál?



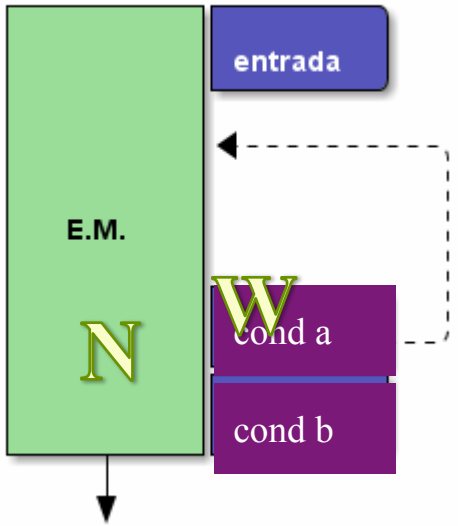
Monitor.- Variantes

- Supongamos que:
  - W espera en cond c
  - N ejecuta c.notify() y reactiva W

- Alternativas de solución (variantes de monitores)
  - El hilo N abandona el monitor (**modelo de Brinch Hansen**)
  - El hilo N espera en una cola especial (**modelo Hoare**)
  - El hilo W espera en la entrada (**modelo Lampson-Redell**)

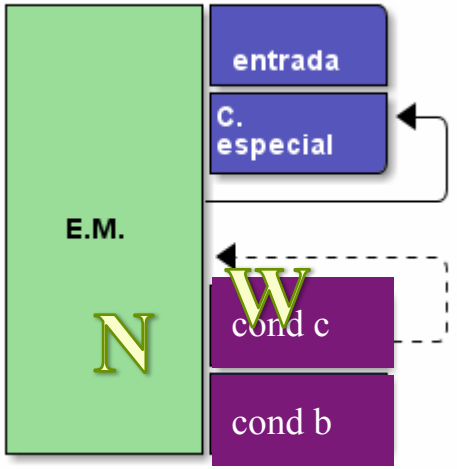
Monitor tipo **Brinch Hansen**

- Supongamos que:
  - W espera en cond c
  - N ejecuta **c.notify()** y reactiva W
- La sentencia ***notify*** es obligatoriamente la última sentencia del método
  - N abandona el monitor y despierta al hilo W
- Cumple exclusión mutua
  - N abandona el monitor
  - W queda activo en el monitor
- No puede aplicarse siempre
  - Algunos problemas complejos requieren realizar otras acciones tras ***c.notify()***
  - No puede aplicarse *notificación en cascada*



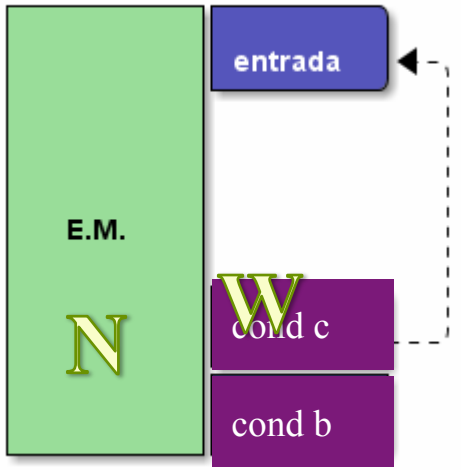
Monitor tipo **Hoare**

- Supongamos que:
  - W espera en cond c
  - N ejecuta ***c.notify()*** y reactiva W
- Además de la entrada, hay una cola especial
  - Para esperar a que el monitor quede libre
  - Prioritaria sobre cola de entrada
- Cuando N ejecuta ***c.notify***
  - N pasa a la cola especial
  - W queda activo en el monitor
- Cumple exclusión mutua
  - N espera fuera del monitor
  - W queda activo en el monitor



Monitor tipo **Lampson-Redell**

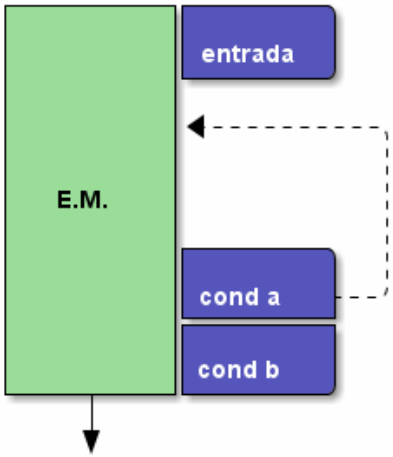
- Supongamos que:
  - W espera en cond c
  - N ejecuta c.notify() y reactiva W
- Cuando N ejecuta *c.notify*
  - W pasa a la cola de entrada
  - N queda activo dentro del monitor
- Cumple exclusión mutua
  - W espera fuera del monitor (en la **entrada**)
    - Cuando consiga entrar, el estado puede haber cambiado de nuevo: hay que reevaluar la condición
  - N queda activo en el monitor



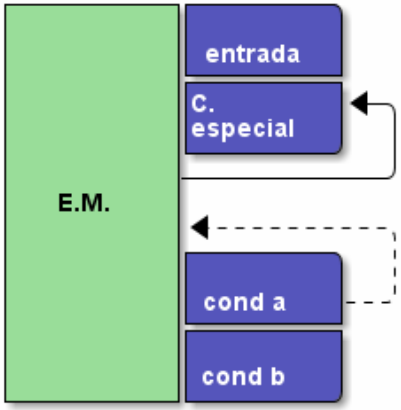
Variantes de monitor.- Resumen

Sigue W (encuentra condición OK)

Brinch Hansen

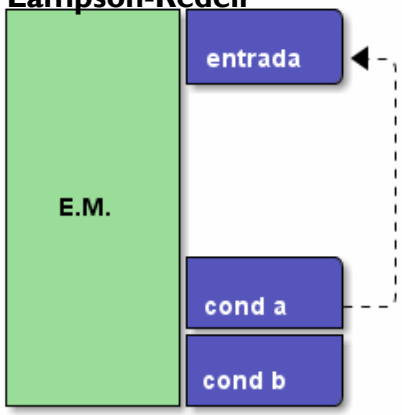


Hoare



Sigue N (W reevaluará condición cuando entre)

Lampson-Redell



Monitor.- Ejemplo hormigas

- El territorio se modela mediante un monitor
  - Atributos:
    - Una matriz de valores lógicos, indicando para cada celda si libre/ocupada
      - `boolean[N][N] ocupada;  // NO es Java`
  - Métodos:
    - *entry void desplaza(x,y,x',y')* La hormiga se desplaza desde la celda (x,y) a (x',y')
- Dos alternativas de solución:
  1. Una cola de espera por celda para esperar a que esa celda quede *libre*
     - `condition[N][N] libre;  // NO es java`
  2. Una cola de espera única denominada *libre*
     - `condition libre;  // NO es java`

Monitor.- ejemplo Hormigas

Alternativa 1

```
Monitor  Terreno {
   boolean[N][N] ocupada;
   condition[N][N] libre;

   entry void desplaza(int x,y,x',y') {
      if (ocupada[x',y'])
           libre[x',y'].wait();
      //actualiza matriz
      ocupada[x',y']=true;
      ocupada[x,y]=false;
      //para avisar a quien quiere ir a x,y
      libre[x,y].notify();
   }
}
```

Alternativa 2

```
Monitor Terreno {
   boolean[N][N] ocupada;
   condition libre;

   entry void desplaza(int x,y,x',y') {
      while (ocupada[x',y'])
           libre.wait(); libre.notify();
      //actualiza matriz
      ocupada[x',y']=true;
      ocupada[x,y]=false;
      //para avisar a quien quiere ir a x,y
      libre.notify();
   }
}
```

Monitor.- Ejemplo hormigas

- Opción 1
  - Mucho más eficiente
    - Sólo reactiva a una hormiga si ha quedado libre la celda por la que espera
  - Uso de *if* sólo posible en algunas variantes de monitor (discutidas luego). El resto requieren *while*

- Opción 2
  - Menos eficiente
    - **Reactivación en cascada**: en el peor caso puede llegar a reactivar a todas las hormigas suspendidas, tras liberar cada celda
  - No se puede aplicar en variante de monitor de Brinch Hansen
  - Única opción si no podemos definir varias colas de espera
- Siempre que resulte posible, elegimos la alternativa 1

Java.- Monitor

- Dos niveles posibles
  - Soporte básico en el lenguaje
  - Soporte extendido (mediante la biblioteca *java.util.concurrent*)
- En esta unidad nos centramos en el soporte básico
  - *java.util.concurrent* se desarrolla en la unidad 5

Java soporta el concepto de monitor

- Todo objeto posee de forma implícita (sin necesidad de declararlos)
  - Un lock
    - Al etiquetar un método con ***synchronized***, se garantiza ejecución en exclusión mutua
      - equivale a *cerrar lock* antes de su primera instrucción y *abrir lock* tras la última
  - Una cola de espera con primitivas
    - **wait()** espera sobre la cola de espera
    - **notify()** reactiva a uno de los hilos que esperan en dicha cola
    - **notifyAll()** reactiva a todos los que esperan
- **Así NO** podemos declarar otros *locks* ni otras colas de espera

Java.- Cómo definir un Monitor

- Una clase que defina objetos a compartir entre hilos, debe:
  - Definir todos sus atributos como **privados**
  - Sincronizar todos sus métodos no privados (***synchronized***)
  - En implementación de cada método, acceder sólo a atributos de la clase y variables **locales**
  - Utilizar *wait(), notify(), notifyAll()* dentro de métodos sincronizados

- OJO.- El compilador no comprueba absolutamente nada (es responsabilidad del programador)
  - No hay ningún tipo de aviso ni error si hay atributos no privados, o métodos públicos sin la palabra *synchronized*, o se utiliza *wait(), notify(), notifyAll()* en un método no sincronizado

Java.- Cómo definir un Monitor (cont.)

- En monitor teórico: los hilos que esperan por condiciones lógicas **distintas** esperan en **colas de espera distintas**
  - Ej.- en productor consumidor podemos tener colas *noVacio*, *noLleno*.
    - Productores que encuentran buffer lleno esperan en *noLleno*
    - Consumidores que esperan porque el buffer está vacío esperan en *noVacio*
- Java utiliza únicamente **una variable condición** por monitor
  - Los hilos que esperan por condiciones lógicas distintas **esperan en una única cola**, no en colas diferentes
  - Al reactivar un hilo no sabemos si reactivamos al que esperaba por una condición o por otra
    - Excepto en casos muy simples, se recomienda despertar a todos y que cada uno vuelva a comprobar su condición
  - La biblioteca *java.util.concurrent* (ver unidad didáctica 5) resuelve esa limitación

Java.- Cómo definir un Monitor (cont.)

- El esquema típico de un método en un objeto compartido en Java es:

```
while (condicionLogica) { // espera mientras condición cierta
        wait(); // try { wait } catch (InterruptedException e) {..}
    }
    ... // código normal
notifyAll(); // si hemos modificado el estado del objeto,
                    avisa a los hilos en espera
```

Java.- Ejemplo hormigas

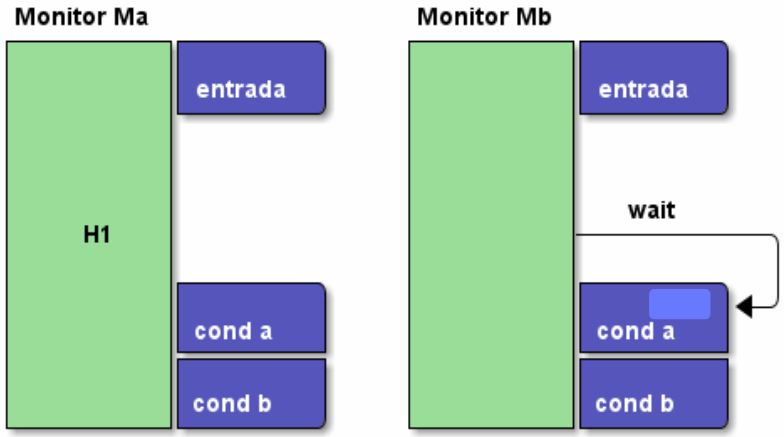
```
public class Territorio {
    private boolean[][] ocupada;

    public Territorio(int N) {
        ocupada=new boolean[N][N];
        for (int i=0; i<N; i++)
          for (int j=0; j<N; j++)
              ocupada[i][j]=false; // libre
    }

    public synchronized void desplaza(int x0, int y0, int x, int y) {
        while (ocupada[x][y])
            try { wait(); } catch (InterruptedException e) {};
        ocupada[x0][y0]=false; ocupada[x][y]=true;
        notifyAll();
    }
}
```

Contenido

- Lenguajes de programación concurrente
- Concepto de monitor
- Variantes de monitor
- Invocaciones cruzadas

Monitor.- invocaciones cruzadas

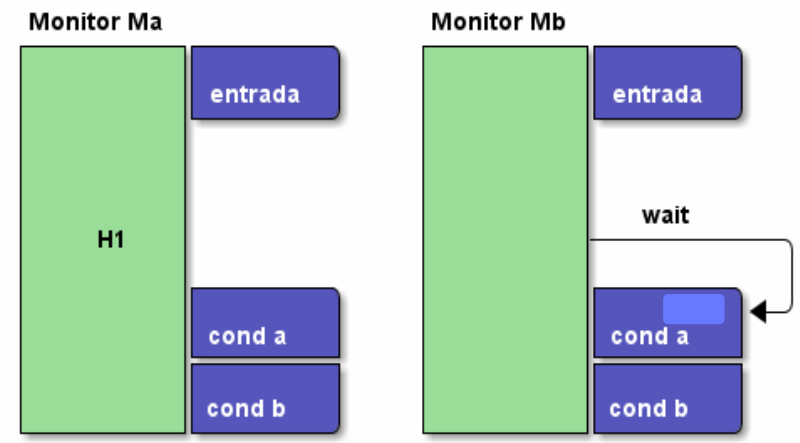
- Invocar desde un monitor a un método de otro monitor puede:
  - reducir la concurrencia
  - incluso provocar interbloqueos



- Ejemplo: Suponemos
  - 2 monitores Ma y Mb:
    - desde un método de Ma se invoca un método de Mb y viceversa
  - 2 hilos H1 y H2

Monitor.- invocaciones cruzadas

- H1 activo en Ma, invoca un método de Mb, dentro del cual ejecuta **a.wait()**
  - Pasa a la cola de espera "a" del monitor Mb
  - Libera el monitor Mb, pero no Ma
    - Nadie puede usar Ma → reducimos concurrencia
- Si H2 entra en Mb (que estaba libre) e invoca un método del monitor Ma
  - Espera en la cola de entrada de Ma (Ma está ocupado)
  - No deja libre el monitor Mb
  - Hemos llegado a un **interbloqueo**



Invocaciones cruzadas.- Ejemplo 1 de interbloqueo

```
public class Problema {
  public synchronized void hola() {...}
        public synchronized void test (Problema x) {
                x.hola();
        }
}
```

- Dos hilos H1, H2
- Dos variables p1, p2

H1 { p1.test(p2) }
H2 { p2.test(p1) }

¿Qué ocurre aquí?

Invocaciones cruzadas.- Ejemplo 2 de interbloqueo

- Definimos dos monitores (p,q) tipo Bcell
- Suponemos 2 hilos concurrentes H1 y H2
  - H1 invoca ***p.swap(q)***, obtiene acceso al monitor p, e inicia la ejecución de *p.swap*
  - H2 invoca ***q.swap(p)***, obtiene acceso al monitor q, e inicia la ejecución de *q.swap*

¿Qué ocurre aquí?

```
class BCell {
  int value;
  public synchronized void getValue() {
    return value;
  }
  public synchronized void setValue(int i) {
    value=i;
  }
  public synchronized void swap(BCell x) {
    int temp= getValue();
      setValue(x.getValue());
      x.setValue(temp);
  }
}
```

Invocaciones cruzadas.- Ejemplo 2 de interbloqueo

- **Aparece un interbloqueo**
  - Dentro de ***p.swap***, H1 invoca ***q.getValue()***, pero debe esperar porque el monitor q no está libre
  - Dentro de ***q.swap***, H2 invoca ***p.getValue()***, pero debe esperar porque el monitor p no está libre
  - Ambos se esperan mutuamente, y la situación no puede evolucionar → **INTERBLOQUEO**

```
class BCell {
  int value;
  public synchronized void getValue() {
    return value;
  }
  public synchronized void setValue(int i) {
    value=i;
  }
  public synchronized void swap(BCell x) {
    int temp= getValue();
      setValue(x.getValue());
      x.setValue(temp);
  }
}
```

Resultados de aprendizaje de la Unidad Didáctica

- Al finalizar esta unidad, el alumno deberá ser capaz de:
  - Programar soluciones eficientes al problema de la sincronización condicional, utilizando monitores.
  - Diseñar adecuadamente un nuevo monitor, en función de las condiciones que deba gestionar.
  - Comparar las variantes de monitor existentes.
    - Clasificar los lenguajes de programación concurrente en función de la variante que soporten.