



Departamento de Programación  
Facultad de Informática  
Universidad Nacional del Comahue



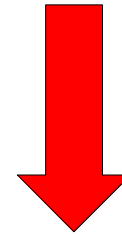
# Programación Concurrente




*Sincronización de cooperación - Monitores*  
*Modelos de Objetos*

# Programación Concurrente

¿Cómo implementar concurrencia en el paradigma orientado a objetos?




Programación Orientada a Objetos  
CONCURRENTE



# Programación Concurrente: modelos de objetos


- *Características:*
  - clase, estado, referencia, método, identidad, encapsulación
- *Operaciones básicas:*
  - Aceptar un mensaje
  - Actualizar el estado
  - Enviar un mensaje
  - Crear nuevos objetos
- *Categorías de objetos*
  - Objeto pasivo
  - Objeto activo



# Programación Concurrente: modelos de objetos

## *Categorías de objetos*


- Objeto pasivo.
  - Espera por un mensaje para invocar una operación (desde un objeto activo)
  - Pueden invocar operaciones sobre otros obj. pasivos
  - No tiene hilo de control, son instancias de clases pasivas
- Objeto activo



# Programación Concurrente: modelos de objetos

## *Categorías de objetos*

- Objeto pasivo.
- Objeto activo.
  - Objeto autónomo que se ejecuta de forma “independiente” de otros objetos activos.
  - Son tareas concurrentes que tienen su propio hilo de control
  - Pueden iniciar acciones que impactan sobre otros objetos




# Programación Concurrente: modelos de objetos

## *Categorías de objetos*

- Objeto pasivo.
- Objeto activo.

Una operación de un objeto pasivo, una vez invocada se ejecuta dentro del hilo de control del objeto activo que la invocó

El modelo concurrente incluye características de las 2 categorías



# Programación Concurrente: modelos de objetos

*Entonces*

- **Aplicación secuencial**
  - Es un programa que consiste de objetos pasivos y tiene solo un hilo de control
- **Aplicación concurrente**
  - Hay varios objetos activos
  - Aparece el concepto de tarea
    - Se debe considerar
      - La capacidad para estructurar la aplicación en tareas concurrentes
      - La capacidad para que las tareas se comuniquen y sincronicen sus operaciones (seguridad y viveza)

# Programación concurrente: Modelos de Objetos

*¿Cómo implementar concurrencia en el paradigma orientado a objetos?*

- se asocian las tareas a los métodos y los hilos que utilizan esas tareas a objetos (activos)
- un sistema es un conjunto de objetos autónomos que colaboran, activa y concurrentemente.
- Modelos de objetos
  - **Modelo activo**
  - **Modelo pasivo**

**Modelo mixto**



# Modelos de objetos

- **Modelo Pasivo**
  - Objetos en modelo secuencial
  - Pasaje de mensajes por invocación de procedimientos
- **Modelo Activo**
  - Cada objeto es autónomo
  - Sistemas orientados a objetos distribuidos
  - Pasaje de mensajes vía comunicación remota
  - Modelo de actores
- **Modelo Mixto** (conurrencia en Java)
  - Objetos pasivos: muestran conciencia de los hilos protegiéndose vía locks u otros mecanismos
  - Objetos activos: hilos, que comparten el acceso a objetos pasivos



# Programación Concurrente: Modelo mixto

## **Programa concurrente**

Colecciones de objetos que se coordinan para resolver un problema

Objetos pasivos

Objetos activos

Objetos controladores



# Programación Concurrente: Modelo mixto

## **Objeto activo**

Implementa código procedural que se ejecuta en hilos

## **Objeto pasivo/reactivo**

Actúa solo en respuesta a un requerimiento recibido desde un objeto activo, y controla las interacciones entre los objetos activos

Diagrama de estados



# Programación Concurrente: Modelo mixto

- Pasos sugeridos para crear un programa concurrente
  - Descripción corta del problema a resolver
  - Identificar objetos y relaciones
    - Especificar si el objeto es activo o pasivo
  - Diseñar los objetos activos
  - Diseñar los objetos pasivos
    - Utilizar diagrama de estados

Automata:

Nodos: estados del objeto

Arcos: métodos a ejecutar, pueden producir cambios de estado



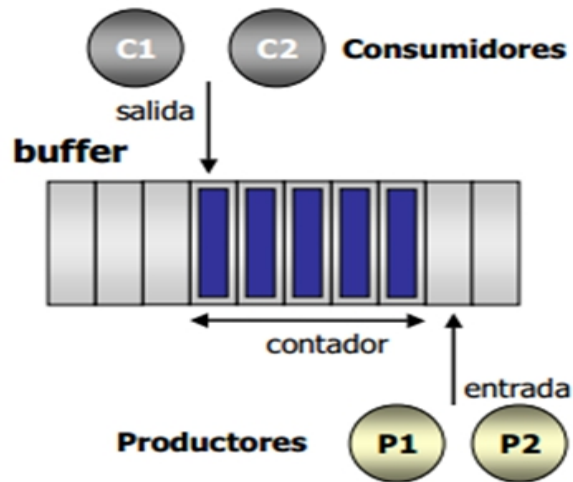
# Programación Concurrente: Modelo mixto

- Pasos sugeridos para crear un programa concurrente
  - Descripción corta del problema a resolver
  - Identificar objetos y relaciones
    - Especificar si el objeto es activo o pasivo
  - Diseñar los objetos activos
  - Diseñar los objetos pasivos
    - Utilizar diagrama de estados
  - Implementar objetos activos como hilos
  - Implementar objetos pasivos – *Thread safe*
  - Implementar objeto controlador

# Programación Concurrente – Modelo Mixto

- Diagrama de estados
  - Se puede estar solo en 1 de los estados en un momento específico
  - Una transición de estado es un cambio provocado por un evento o una acción sobre el objeto
  - Puede que no se produzcan cambios
  - El próximo estado depende de la acción aplicada y del estado actual
  - Varias aplicaciones son altamente dependientes del estado. Sus acciones dependen no solo de las entradas sino también de lo ocurrido previamente

# Problema del Productor/Consumidor



Un *productor* produce algún producto (datos, mensaje, ...) que se coloca en una estructura compartida para que sea consumido por un *consumidor*

Casos:

- 1 productor / 1 consumidor
- 1 productor / varios consumidores
- varios productores / 1 consumidor
- varios productores / varios consumidores

# Problema del Productor/Consumidor

- El productor produce en cualquier momento
- El consumidor puede tomar un dato sólo cuando hay disponibilidad
- Para el intercambio de datos se usa una estructura a la cual ambos tienen acceso. (la estructura elegida depende del problema particular a resolver)
- Todo lo que se produce debe ser consumido

Acceso de los productores y consumidores de forma de asegurar consistencia de la información almacenada en la estructura.

Casos:

- La estructura tiene capacidad limitada:
  - a) un productor no puede poner un elemento si no hay lugar,
  - b) un consumidor no puede extraer un elemento si la estructura está vacía
- La estructura tiene capacidad ilimitada:
  - a) un productor siempre puede poner un elemento, siempre hay lugar,
  - b) un consumidor no puede extraer un elemento si la estructura está vacía
- La estructura es de capacidad nula ( + ➡ )

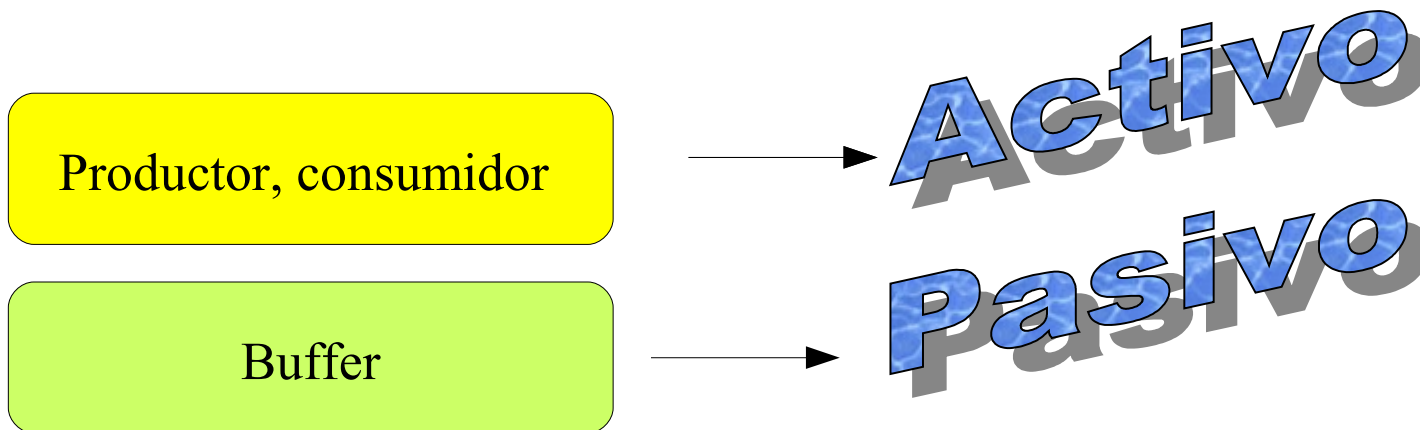


# Programación Concurrente: Modelo mixto

## *Problema del productor/consumidor con un número finito de lugares*

Descripción corta del problema: crear un productor que cree datos y un consumidor que use esos datos y se coordinen para ello utilizando un buffer limitado.

Se debe considerar objetos productor, consumidor y buffer.





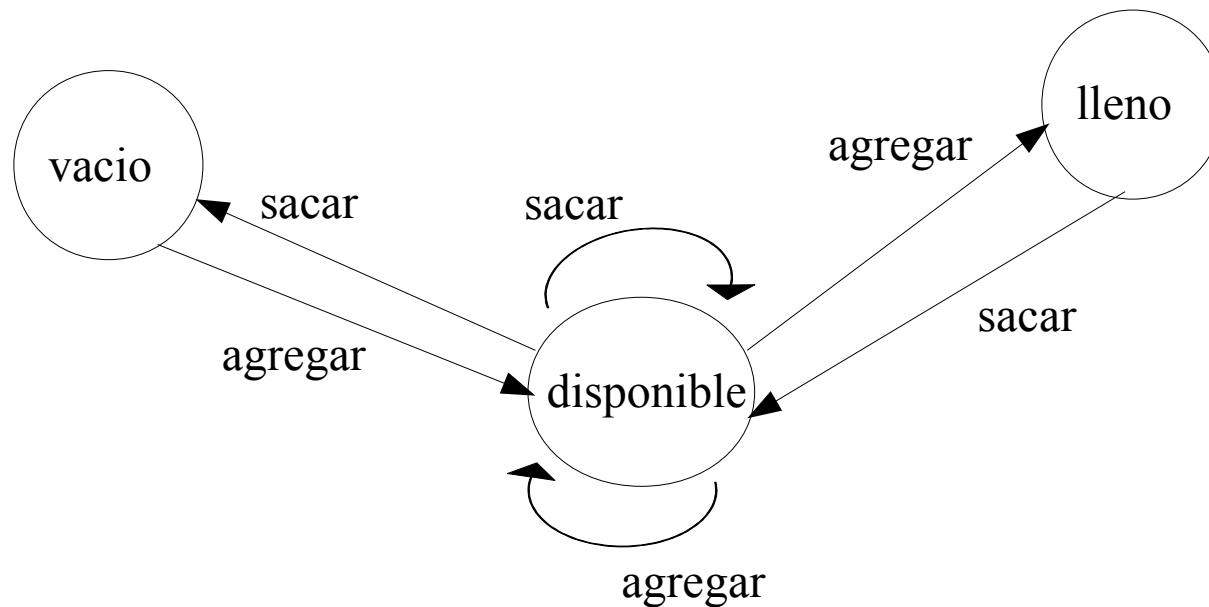
# Programación Concurrente: Modelo mixto

## *Problema del productor/consumidor*

- El buffer responde a 2 tipos de requerimientos: tomar un ítem del productor y agregarlo a su estructura, y tomar un ítem de su estructura y entregarlo al consumidor.
- Entonces, el buffer tendrá 2 métodos: *agregar* y *sacar*.
- Se propone hacer una tabla indicando
  - Nombre del método
  - Estados en que puede ejecutarse
  - Precondición donde se verifique el estado para decidir si corresponde “esperar” (a que se de la condición)
  - Postcondición donde se decide si se actualiza el estado para notificar el cambio

# Programación Concurrente: Modelo mixto

*Problema del productor/consumidor – buffer – diagrama de estado*



# Sincronización competencia y cooperación

## Mecanismos

Monitores + esperas guardadas

Semáforos binarios y generales

Cerrojos + Variables de condición

Otras posibilidades ...



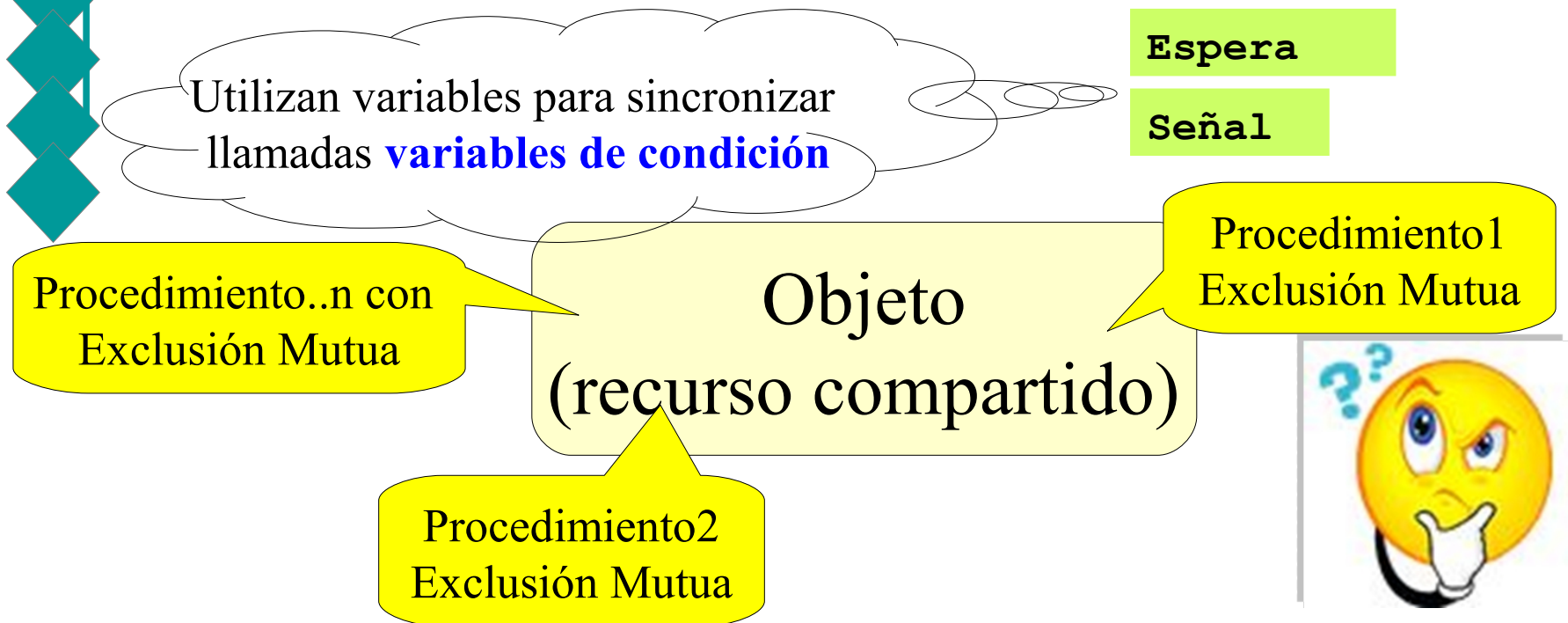
# Sincronización - Monitores

- Un monitor es un mecanismo de abstracción de datos que se aplica en un ambiente concurrente
- Un monitor asegura exclusión mutua
- Un monitor es un objeto pasivo, que generalmente es un recurso compartido
- Un proceso que invoca un método sobre un monitor puede ignorar cómo el método es implementado. Todo lo que importa son los efectos visibles de la invocación
- El programador de un monitor puede ignorar cómo y dónde los métodos del monitor son usados.

# Sincronización - Monitores

- Mecanismo de sincronización de **alto nivel**
  - garantiza que solamente pueda existir un hilo activo dentro de la sección crítica.
- Un monitor **permite suspender un hilo dentro de la sección crítica** posibilitando que otro hilo pueda acceder a la misma.
- Este segundo hilo puede abandonar el monitor, liberándolo, o suspenderse dentro del monitor.
- De cualquier modo, el hilo original se despierta y continúa su ejecución dentro del monitor.
- Este esquema es escalable a múltiples hilos
  - ( varios hilos pueden suspenderse dentro de un monitor).

# Sincronización - Monitor



- Los **procedimientos** van encapsulados dentro de un módulo.
- Dentro de cada **módulo** sólo un proceso puede estar activo cada vez
- Si el monitor se ha codificado correctamente no puede ser utilizado incorrectamente por ningún proceso que desee usar el recurso.

# Sincronización - Monitores



- En Java ... la semántica de monitor se logra por la combinación de “synchronized” y los métodos:  
*wait()*, *notify()*; *notifyAll()*
- Cada objeto tiene un *lock* y un *conjunto de espera* (CE)
- Cualquier objeto puede servir como *monitor*. Aunque cada lenguaje define sus propios detalles
- Cada CE mantiene los hilos bloqueados por un *wait()* sobre el objeto
- Los CE interactúan con los locks, entonces *wait()*, *notify()* y *notifyAll()* requieren sincronización.



# Sincronización - Monitores

*Esperas guardadas* (dependen de una condición)

- En general las invocaciones de `wait()` se hacen dentro de un bucle `while`.
- Cuando una acción es reasumida, la tarea en espera NO SABE si la condición por la que estaba esperando es verdadera.
- Solo sabe que lo liberaron, y debe volver a verificar, para mantener la propiedad de seguridad.
- Es una buena práctica que este estilo se utilice aun cuando haya una única instancia que pueda esperar por la condición.

# Sincronización - Monitor

- No es posible señalar a un hilo en especial.
  - Es aconsejable bloquear a los hilos en el CE con una condición de guarda en conjunción con `notifyAll()`.

*while (!condicion)*

*try {wait();}*

*catch (InterruptedException e) {return;}*

- Todos son despertados, comprobarán la condición y volverán a bloquearse excepto los que la encuentren verdadera .
- **No es aconsejable** comprobar la condición de guarda con **if**
- Los **campos protegidos** por el monitor deben declararse **private**



# Semántica del monitor



Cuando un método `synchronized` del monitor ejecuta un `wait()` libera la exclusión mutua sobre el monitor y el hilo corriente es ubicado en el CE del objeto.

Cuando desde otro método del monitor se ejecuta un `notify()`, un hilo del CE pasará al conjunto de hilos que esperan el cerrojo y se reanudará cuando sea planificado.

Cuando desde otro método del monitor se ejecuta `notifyAll()`, todos los hilos del CE pasarán al conjunto de hilos que esperan el cerrojo y se reanudarán cuando sean planificados.

# Cómo utilizar el método *wait()*

- Utilizar dentro de un método *synchronized* y dentro de un ciclo indefinido que verifica la condición:

```
synchronized void hacerCondicion() {  
    while (!Condicion)  
        this.wait();  
}
```

- Al ejecutar un **wait** sobre un objeto
  - El hilo se suspende, y es ubicado por la JVM en el CE del objeto.
  - Libera el lock** (de sincronización) del objeto.
  - Si el hilo mantiene otros locks en su poder, NO los libera!!!

**Cuidado!!**

# Cómo usar el método *notify()*

- Utilizar dentro de un método *synchronized* :

```
synchronized void cambiaCondicion() {  
    ...//algo cambia en el estado del objeto  
    this.notifyAll() ;  
}
```

- Muchos thread pueden estar esperando sobre el objeto:
  - Con **notify()** solo un thread (no se sabe cual) es despertado.
  - Con **notifyAll()** todos los thread son despertados y cada uno decide si la notificación le afecta, o si no, vuelve a ejecutar el wait().  
(podría suceder que la condición que espera aun no se cumpla)



# Cómo usar el método *notify()*

- El hilo *H* suspendido (por efecto de un `wait()`) **SIEMPRE debe esperar (al menos)** hasta que el hilo que invoca ***notify()*** o ***notifyAll()*** libere el lock del objeto.
- *H* debe volver a obtener el lock del objeto, entonces puede ocurrir que se bloquee si algún otro hilo lo obtiene primero.
- Cuando *H* logra obtener el lock retoma desde el punto de su `wait()`
- Si el CE está vacío `notify()` y `notifyAll()` no tienen efecto

# Mecanica de monitores

Consideremos el siguiente ejemplo inutil

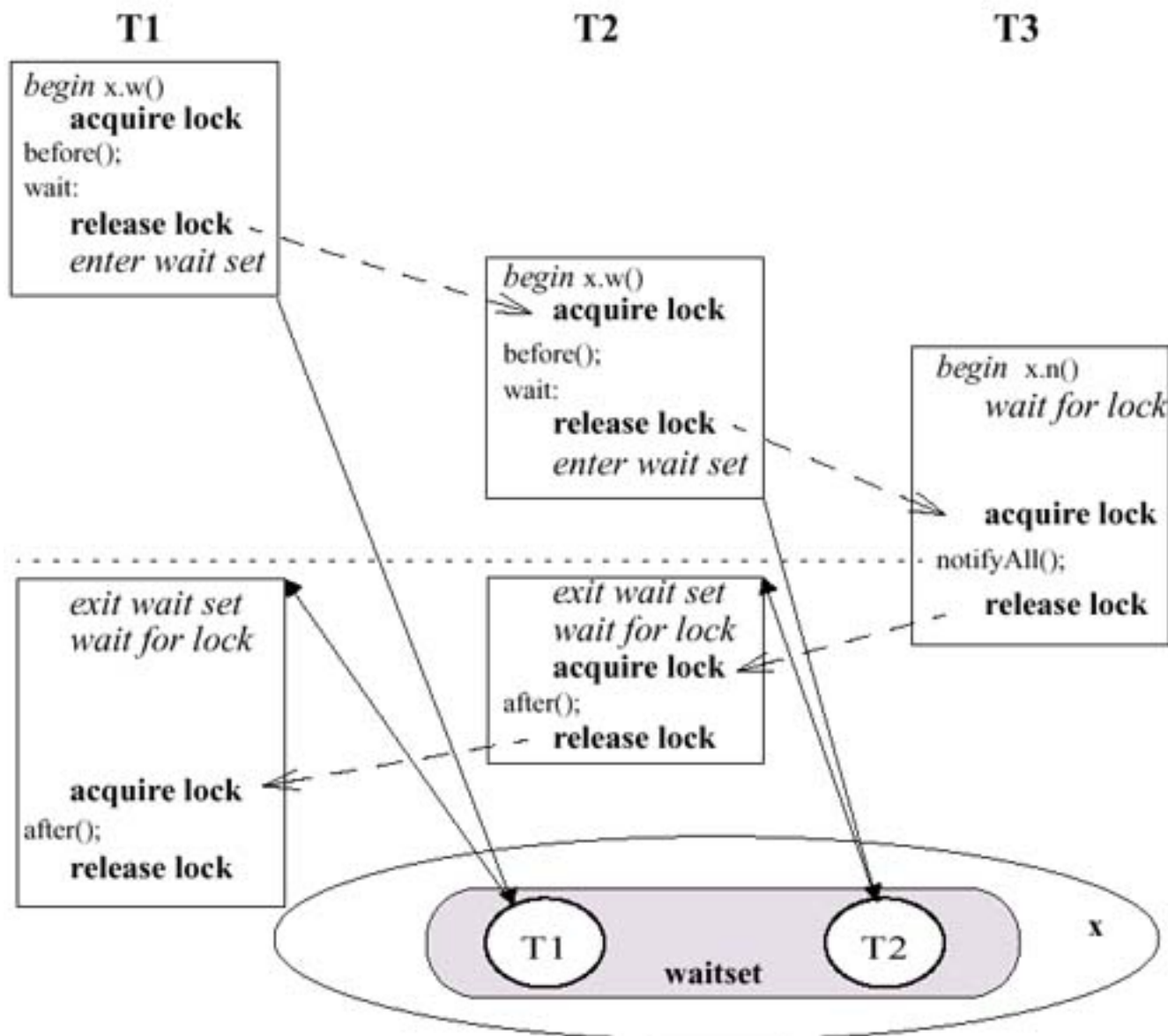
```
class MonX {
    synchronized void w() throws InterruptedException {
        this.before();
        this.wait();
        this.after();
    }

    synchronized void n() {
        notifyAll();
    }

    void before() {}
    void after() {}
}
```

Y consideremos 3 hilos que invocan los métodos del monitor MonX sobre una instancia compartida x, (x instancia de MonX)

Ejemplo tomado de: “*Concurrent Programming in Java*” - Doug Lea







# Volviendo al problema del productor/consumidor

- Tendremos ...
- un hilo productor (o varios)
- un hilo consumidor (o varios)
- un buffer, que será implementado de varias formas, utilizando distintos mecanismos de sincronización
- ...será un monitor ...
- ... o ... semáforos?



A trabajar!!