

# Sesión 11: *Recursos compartidos*

Concurrencia

---

Ángel Herranz

2019-2020

Universidad Politécnica de Madrid



# Productor/consumidor 1 dato

```
// Buffer compartido de 1 dato  
volatile public static Producto p = null;
```

```
Producto tmp;  
while (true) {  
    tmp = Fabrica.fabricar();  
    // Almacenar:  
    < AWAIT p == null →  
        p = tmp; >;  
}
```

```
Producto tmp;  
while (true) {  
    // Extraer:  
    < AWAIT p != null →  
        tmp = p; p = null; >;  
    Consumo.consumir(tmp);  
}
```



# Ejercicio obligatorio semanal

Hoja de ejercicios en:

<http://babel.ls.fi.upm.es/teaching/concurrencia>

Ejercicio 5:

**Almacén de un dato con semáforos**

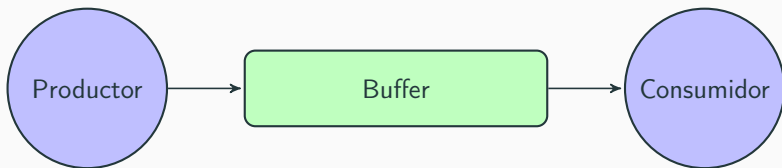
Fichero a entregar:

`Almacen1.java`

Sistema de entrega:

<http://vps142.cesvima.upm.es>

# Arquitectura CC\_05\_P1CSem



- Estudiar bien la **arquitectura** del código
- Procesos **productores**: fabrican y almacenan
- Procesos **consumidores**: extraen y consumen
- **Comparten** el almacén (instancia de **Almacen1**)
- **Encapsular sincronización** en **Almacen1**

# main

```
public static void main(final String[] args) {  
    Almacen almac = new Almacen1();  
    Productor[] productores = new Productor[N_PRODS];  
    Consumidor[] consumidores = new Consumidor[N_CONSS];  
    for (int i = 0; i < N_PRODS; i++)  
        productores[i] = new Productor(almac);  
    for (int i = 0; i < N_CONSS; i++)  
        consumidores[i] = new Consumidor(almac);  
    for (int i = 0; i < N_PRODS; i++)  
        productores[i].start();  
    for (int i = 0; i < N_CONSS; i++)  
        consumidores[i].start();  
    ...  
}
```

# es.upm.babel.cclib.Productor

```
public class Productor extends Thread {  
    private Almacen almacenCompartido;  
    public Productor(Almacen a) {  
        almacenCompartido = a;  
    }  
    public void run() {  
        Producto p;  
        while (true) {  
            p = Fabrica.producir();  
            almacenCompartido.almacenar(p);  
        }  
    }  
}
```

# es.upm.babel.cclib.Consumidor

```
public class Consumidor extends Thread {  
    private Almacen almacenCompartido;  
    public Consumidor(Almacen a) {  
        almacenCompartido = a;  
    }  
    public void run() {  
        Producto p;  
        while (true) {  
            p = almacenCompartido.extraer();  
            Consumo.consumir(p);  
        }  
    }  
}
```

# Almacen1.java

```
class Almacen1 implements Almacen {  
    // Variable compartida: sincronizar por mutex  
    private Producto almacenado = null;  
    public void almacenar(Producto producto) {  
        // Sincronizar por condición: esperar hasta null  
        almacenado = producto;  
    }  
    public Producto extraer() {  
        Producto result;  
        // Sincronizar por condición: esperar hasta no null  
        result = almacenado;  
        almacenado = null;  
        return result;  
    }  
}
```



# Productor/consumidor $n$ datos

```
volatile public static Cola c = new Cola(n);
```

```
Producto p;  
while (true) {  
    p = Fabrica.fabricar();  
    // Almacenar:  
    < AWAIT c.huecos() > 0 →  
        c.almacenar(p); >;  
}
```

```
Producto p;  
while (true) {  
    // Extraer:  
    < AWAIT c.datos() > 0 →  
        p = c.primer();  
        c.borrar(); >;  
    Consumo.consumir(p);  
}
```

Ahora el productor podría fabricar hasta  $n$  productos antes de tener que pararse porque el consumidor fuera lento, y viceversa



# Ejercicio obligatorio semanal

Hoja de ejercicios en:

<http://babel.ls.fi.upm.es/teaching/concurrencia>

Ejercicio 6:

**Almacén de varios datos con semáforos**

Fichero a entregar:

`AlmacenN.java`

Sistema de entrega:

<http://vps142.cesvima.upm.es>



 Simultaneidad

+

Sincronización<sup>1</sup> +  Comunicación<sup>2</sup>

---

<sup>1</sup>Exclusión mutua + sincronización por condición

<sup>2</sup>Sólo con memoria compartida.



# Concurrencia

 Simultaneidad

+

 Sincronización<sup>1</sup> +  Comunicación<sup>2</sup>

---

<sup>1</sup>Exclusión mutua + sincronización por condición

<sup>2</sup>Sólo con memoria compartida.

# Sincronización

Sincronización		
Mutex	por condición	Dificultad
<input checked="" type="checkbox"/>	<input type="checkbox"/>	Fácil
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Difícil
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Muy difícil

- Da igual si estamos en memoria compartida o en paso de mensajes, con diferencia, **la sincronización es lo más difícil** en concurrencia

## Método

1. Detectar procesos
2. Detectar recursos compartidos
3. Especificar los recursos formalmente:  
Estado + Operaciones + Sincronización
4. Razonar a *alto nivel*
5. Generar código siguiendo patrones

# Recursos compartidos: especificación formal

---

# Contexto: métodos formales

- Técnicas **matemáticamente rigurosas** para especificar, desarrollar y verificar software
- Usados para construir software **fiable y robusto**
- **Independencia del lenguaje** de programación
- Para **profundizar**: Z, VDM, B, etc.
- Interesante **colección de herramientas**: rise4fun
- Libro muy recomendable:  
Specifying Systems: The TLA+ Language and  
Tools for Hardware and Software Engineers

*Leslie Lamport*

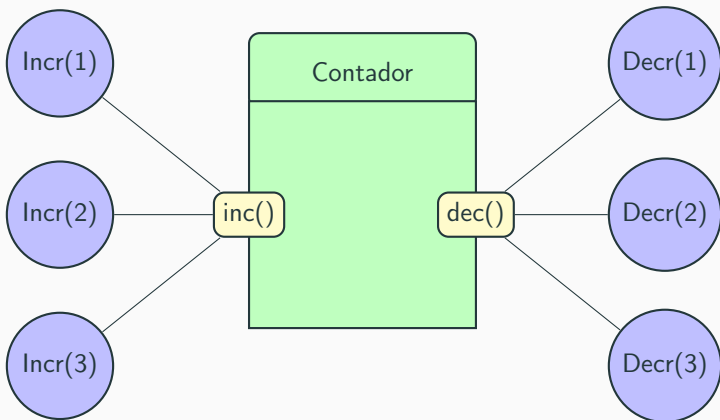
<https://lamport.azurewebsites.net/tla/book.html>



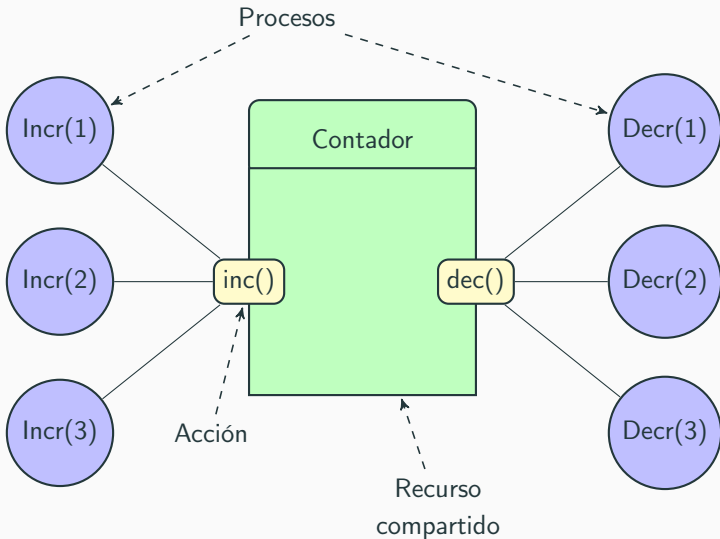
# Razonar a *alto nivel*

- Todas las **interacciones** entre procesos se realizan a través de recursos compartidos
- Los recursos definen acciones atómicas
- El efecto de cada acción es global:  
*accesible por todos los procesos*
- El efecto de dichas acciones es serializable:  
*el efecto de que dos procesos ejecuten dos acciones  $A_1, A_2$  es equivalente a uno de los entrelazados  $A_1 \rightarrow A_2$  o  $A_2 \rightarrow A_1$*

# Diagrama de procesos y recursos



# Diagrama de procesos y recursos



# Recursos compartidos: sincronización

- **Exclusión mutua**: se deriva **automáticamente** de la atomicidad y serializabilidad de las acciones
- **Sincronización por condición**: se especificará **explícitamente** e impondrá **restricciones a los entrelazados** de las acciones

# Ejemplo de especificación: Contador

**C-TAD** Contador  
**OPERACIONES**  
ACCIÓN inc:  
ACCIÓN dec:

**SEMÁNTICA**

**DOMINIO**  
**TIPO** Contador =  $\mathbb{N}$

**INICIAL** self = 0

**CPRE** cierto  
inc()  
**POST** self = self<sup>pre</sup> + 1

**CPRE** self > 0  
dec()  
**POST** self<sup>pre</sup> = self + 1

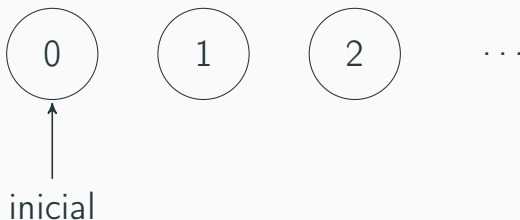
- **C-TAD**: TAD concurrente
- **OPERACIONES**: sección de **sintaxis**
- **ACCIÓN**: acciones atómicas del recurso, en este caso inc y dec sin parámetros
- **SEMÁNTICA**: sección de **comportamiento**
- **DOMINIO** y **TIPO**: representación interna de la información encapsulada en el recurso: un número natural
- **INICIAL**: valor inicial del recurso
- **CPRE**: sincronización por condición
- **POST**: relaciona el estado después de ejecutar la acción con el estado antes

# DOMINIO, TIPO, INICIAL y self

- **self** es el **valor concreto** del estado del recurso en un **determinado instante**
- **self** puede tomar valores en el **conjunto** especificado por **DOMINIO**
- En Contador, **self** puede ser cualquier valor de  $\mathbb{N}$ : 0, 1, 2, 3, ...
- **self** *es como* **this** en Java o **self** en Python
- **INICIAL** es un **predicado** que describe un posible **estado de partida** del recurso

# Recursos como autómatas

- En muchas ocasiones vamos a usar autómatas para representar y razonar sobre la evolución de un recurso compartido
- Así son los posibles estados del recurso Contador



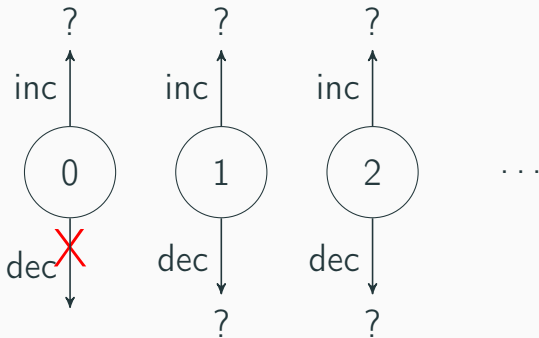
# CPRE: sincronización condicional

- Cada acción tiene una **CPRE**: **predicado** que establece si el **estado** del recurso es **apropiado** para la acción
- Si no lo es, **el proceso** que intenta ejecutarla **espera** hasta que el predicado se cumpla
- Volviendo a los autómatas: si la **CPRE** es cierta para un estado del recurso entonces hay una **transición habilitada**



# CPRE en el autómata

En la acción dec  
**self** = 0  $\Rightarrow \neg$ CPRE



# POST: antes-despues

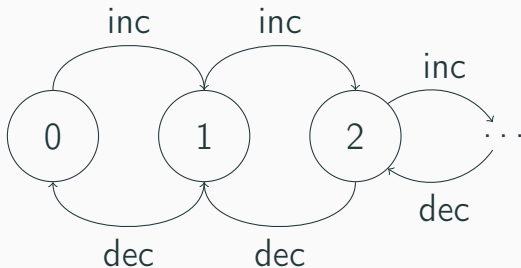
- Cada acción tiene una **POST**: **predicado** que especifica el efecto de una acción mediante la **relación entre los estados antes y despues** de ejecutar la acción
- En el ejemplo Contado la **POST** de dec dice

$$\mathbf{self}^{\text{pre}} = \mathbf{self} + 1$$

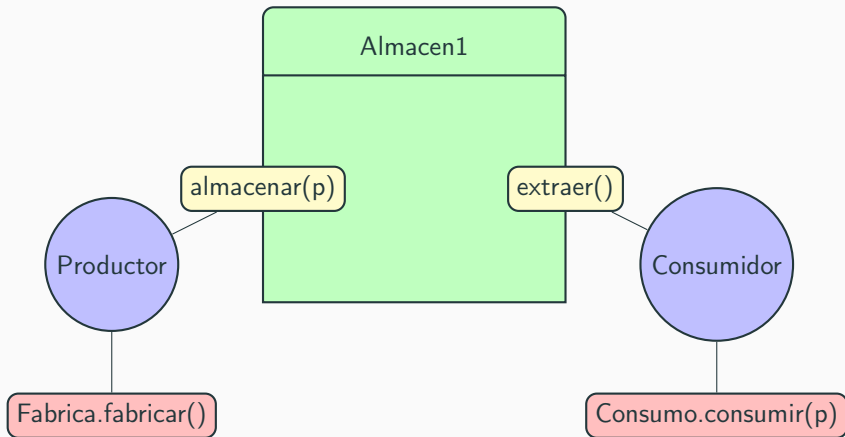
i.e. si el estado del recurso **antes** de ejecutar era  $A + 1$  entonces el estado del recurso **despues** de ejecutar dec es  $A$

# POST en el autómata

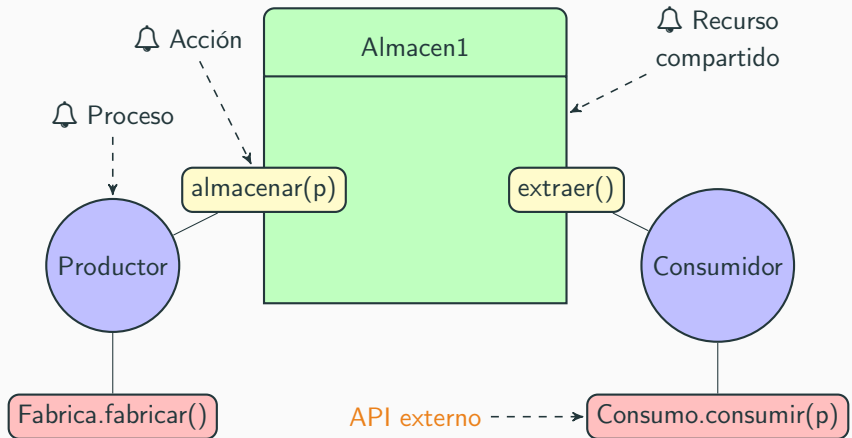
Las **POSTs** definen las transiciones entre estados del autómata



# Productor/Buffer/Consumidor



# Producer/Buffer/Consumer



# Especificación de Almacen1

C-TAD Almacen1

## OPERACIONES

ACCIÓN almacenar: Producto[e]

ACCIÓN extraer: Producto[s]

## SEMÁNTICA

### DOMINIO

TIPO Almacen1 =  $(p : \text{Producto} \times \text{hay\_dato} : \mathbb{B})$

# Especificación de Almacen1

INICIAL:  $\neg \text{self.hay\_dato}$

CPRE:  $\neg \text{self.hay\_dato}$

    almacenar(p)

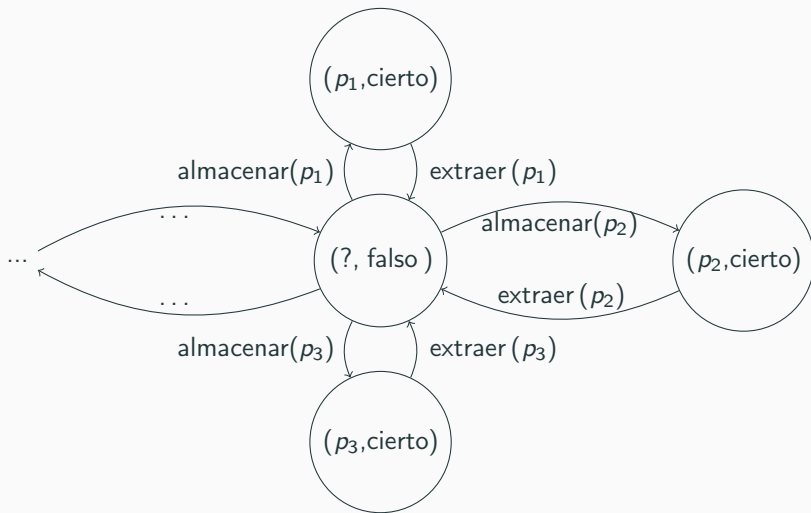
POST:  $\text{self.hay\_dato} \wedge \text{self.p} = p^{\text{pre}}$

CPRE:  $\neg \text{self.hay\_dato}$

    extraer(p)

POST:  $\neg \text{self.hay\_dato} \wedge p = \text{self.p}^{\text{pre}}$

# Autómata de Almacen1





## **Especificación de recursos para programación concurrente**

Manuel Carro      Julio Mariño

Ángel Herranz

Dpto. de Lenguajes, Sistemas Informáticos e Ing. de Software  
Universidad Politécnica de Madrid

Revisión SVN 2027 / 2015-03-26

### **Índice**

<b>1. Razones para especificar</b>	<b>2</b>
<b>2. Especificación de recursos</b>	<b>3</b>
2.1. Declaración de nombre y de operaciones . . . . .	4
2.2. Declaración de dominio . . . . .	5
2.3. Estado inicial del recurso	7

Disponible en

<http://babel.ls.fi.upm.es/teaching/concurrencia/>