# Asignación de Prácticas Número 8

## Programación Concurrente y de Tiempo Real

Antonio J. Tomeu[1]

[1]Departamento de Ingeniería Informática
Universidad de Cádiz

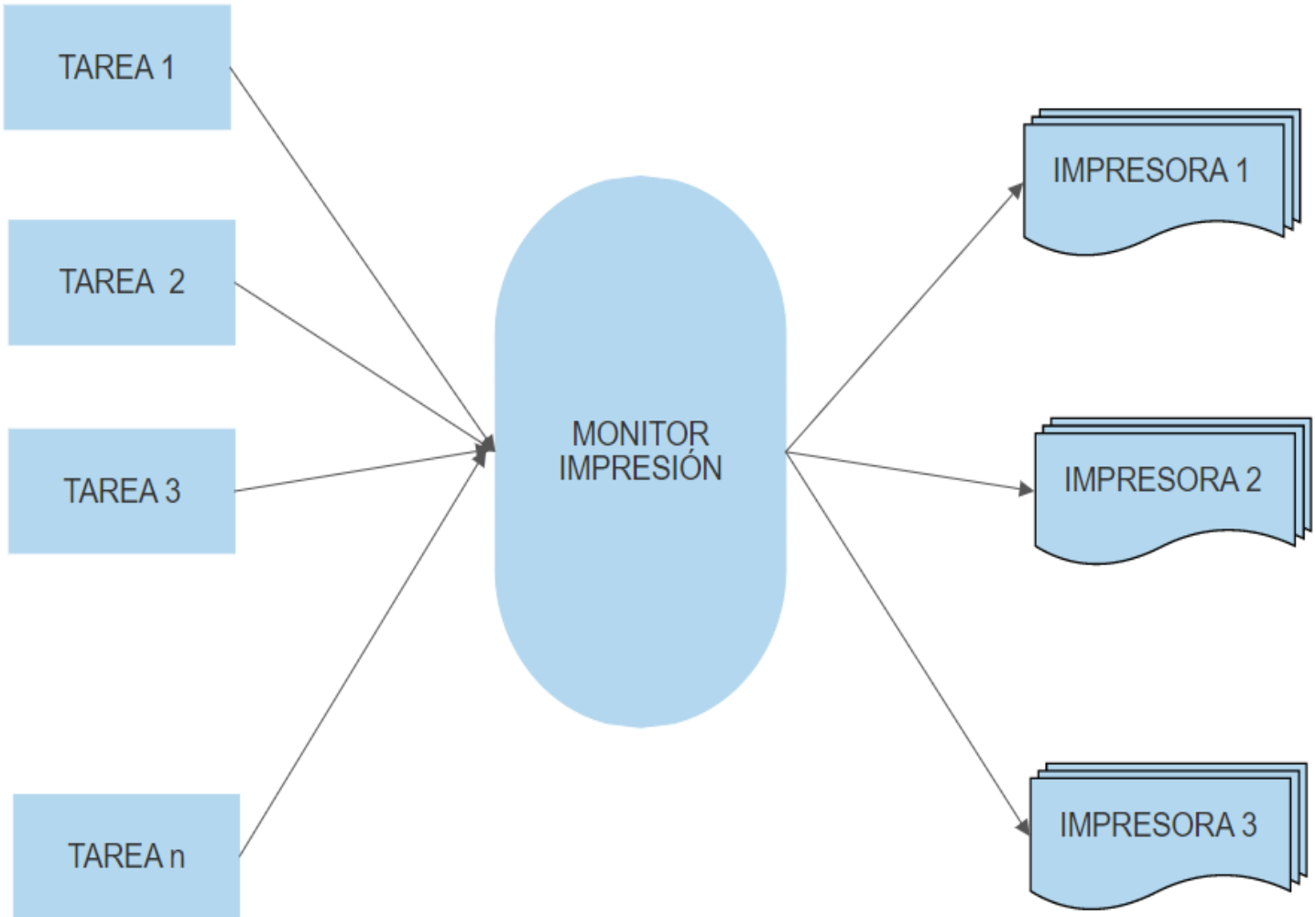
PCTR, 2020

# Contenido I

Objetivos de la Práctica

- Los mismos que para la práctica anterior.

# Trabajamos en el Ejercicio Número 1 I



# Trabajamos en el Ejercicio Número 1 I

- Se trata de escribir un monitor que arbitre el acceso de tareas impresoras a un sistema de tres impresoras.
- Nos preguntamos:
  - Pregunta: qué datos encapsulamos en el monitor?
  - Respuesta: un array booleano llamado `libres`. Cada ranura del array representará el estado de una de las impresoras del sistema (libre u ocupada). También será neceario un entero `impresoras` que indique cuántas impresoras están libres.
  - Pregunta: cómo será el código de inicialización del monitor:
  - Respuesta: muy sencillo; ajustará el valor de impresoras a 3 (todas libres) y todas las ranuras del array a (libres).
  - Pregunta: qué procedimientos (métodos) tendrá el monitor?

# Trabajamos en el Ejercicio Número 1 II

- Respuesta: dos; `procedure integer take_print()`, donde una tarea impresora pide al monitor imprimir, y recibe como respuesta un entero que le indica qué numero de impresora debe utilizar, y `procedure drop_print(var n: integer)`, donde una tarea que ha terminado de imprimir indica al monitor que deja libre la impresora, indicándola mediane el parámetro `n`.
- Pregunta: qué sincronización necesita el monitor?
- Respuesta: necesitará dormir a aquellas tareas impresoras que encuentren que todas las impresoras están ocupadas al pedir el procedimiento `take_print()`. Cuando una impresora `i` quede libre vía método `drop_print(i)`, desde el mismo se deberá despertar a alguna de las tareas impresoras que estaban bloqueadas a la espera de impresoras libres.
- Pregunta: cómo implanto la sincronización descrita?
- Respuesta: fácilmente, con una variable de condición `imp: condition`.

# Trabajamos en el Ejercicio Número 1 III

- Presentamos un esqueleto del monitor que hay que desarrollar, y que debe completar:

```
1  monitor impresoras;
2    var
3      i, impresoras: integer;
4      libre: array[1..3] of boolean;
5      imp: condition;
6
7    procedure integer take_print();
8    begin
9      (*completar*)
10   end;
11
12   procedure drop_print(var n: integer);
13   begin
14     (*completar*)
15   end;
16
17   begin
18     for i:=1 to 3 do
19       libre[i]:=true;
```

# Trabajamos en el Ejercicio Número 1 IV

```
20      impresoras:=3;
21    end.
```

- Pregunta: qué estructura tendrá las tareas impresoras que usan el monitor:
- Respuesta: algo como lo siguiente:

```
1  task impresoras is
2    i: integer;
3  begin
4    while true
5      begin
6        i:=take_print();
7        imprimir en i;
8        drop_print(i);
9      end;
10 end;
```

- Pregunta: qué hago ahora?

# Trabajamos en el Ejercicio Número 1 V

- Respuesta: a partir de lo expuesto, derivar el código de un monitor teórico, y obtener su equivalente en Java aplicando el protocolo de diseño de monitores.

# Trabajamos en el Ejercicio Número 2 I

- Se trata de implementar un monitor en Java que de soporte al problema de los filósofos.
- Este problema ya le ha sido presentado en clase de teoría...
- ... pero lo repasamos brevemente.
- Para ello, en la carpeta de la práctica dispone de un documento `filosofos.pdf`, con la solución al problema mediante monitores descrita en la segunda edición del libro de Ben-Ari. Ahora, pare el vídeo, y repase el texto.
- Hay una simulación visual del problema que puede ayudar en `https://www.youtube.com/watch?v=H33eWKOiUJE`
- Analizamos a continuación el monitor que propone este autor para resolver el problema (hay otras aproximaciones en la literatura).

# Modelo Teórico Para El Problema de los Filósofos I

```
monitor ForkMonitor
  integer array[0..4] fork ← [2, . . . , 2]
  condition array[0..4] OKtoEat
  operation takeForks(integer i)
    if fork[i] ≠ 2
       waitC(OKtoEat[i])
    fork[i+1] ← fork[i+1] - 1
    fork[i-1] ← fork[i-1] - 1
  operation releaseForks(integer i)
    fork[i+1] ← fork[i+1] + 1
    fork[i-1] ← fork[i-1] + 1
    if fork[i+1] = 2
       signalC(OKtoEat[i+1])
    if fork[i-1] = 2
       signalC(OKtoEat[i-1])
```

**philosopher i**

```
    loop forever


p1:     think
p2:     takeForks(i)
p3:     eat
p4:     releaseForks(i)
```

# Qué Hago Ahora? I

- Asegúrese de entender bien la solución de Ben-Ari.
- Aplique el protocolo de diseño de monitores, y obtenga un equivalente funcional en Java.
- Atención: el monitor teórico da al array `fork` estructura circular. Asegúrese de hacer lo mismo en Java, controlando aquellos casos donde `i+1` e `i-1` se salen de los límites del array.
- Finalmente, escriba un programa que crea cinco tareas filósofo y sincronícelas utilizando el monitor.