Programación Concurrente

Hernán Melgratti

hmelgra@dc.uba.ar

Condiciones generales

- Horario: Viernes de 9:00 a 13:00.
- Clases teórico/prácticas
  - Resolución de problemas en laboratorio
- Evaluaciones:
  - Problemas que se resolverán durante el dictado
  - Un parcial
  - Un trabajo práctico final

Concurrencia vs Paralelismo

- Paralelismo
  - Ocurre físicamente al mismo tiempo
- Concurrencia
  - Ocurre lógicamente al mismo tiempo, pero podría ser implementado sin paralelismo real

Objetivos

- Introducir a problemas comunes a muchas disciplinas
  - Sistemas operativos
  - Sistemas distribuidos
  - Sistemas en tiempo real
- Comprender problemas clásicos de la programación concurrente
  - Problemas clásicos de sincronización

Objetivos

- Comprender las principales primitivas para la programación concurrente
- Desarrollar habilidades para utilizar estas primitivas para resolver problemas de sincronización
- Conocer técnicas de programación de lenguajes de programación concurrentes modernos

Material de estudio

- Transparencias de clases
- Bibliografía:
  - Doug Lea, Concurrent Programming in Java™: Design Principles and Patterns, Addison Wesley, 1999
  - M. Ben-Ari, Principles of Concurrent and Distributed Programming, Addison-Wesley, 2006.

Introducción

- Introducción a la programación concurrente
- Conceptos básicos
  - Principios de concurrencia
    - Procesos/Threads
    - Estado, Ejecución, *Scheduling*
  - Problemas de sincronización
- Introducción a lenguajes de programación
  - Java

¿Por qué concurrencia?

- Tradicionalmente
  - Utilizar el procesador eficientemente ante la presencia de operaciones de entrada/salida
    - Sistemas operativos
    - Sistemas distribuidos
    - Sistemas en tiempo real
- Modelar sistemas inherentemente concurrente
  - Controladores de software que tienen que responder a varios sensores físicos

Problema

- Escribir un programa java que muestre el mensaje “Hola” cada 3 segundos
- Para esperar


```
try {
    Thread.sleep(3000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Solución

```
public class Hola {
	public static void main(String args[])
	{
		while (true){
			System.out.println("Hola");
			try {
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Problema II

- Modificar el programa para que muestre también el mensaje "Como estás" cada 5 segundos

Solución

```
public class HolaComoEstas {
   public static void main(String args[]) {
                    int proxA = 3000;
                    int proxB = 5000;
                    int esp;
                    while (true) {
                              if (proxA<= proxB) {
                                        System.out.println("Hola");
                                        proxB = proxB-proxA;
                                        proxA = 3000;
                              } else{
                                        System.out.println("Como estas");
                                        proxA = proxA-proxB;
                                        proxB = 5000;
                              }
                              if (proxA<proxB){
                                        esp = proxA;
                              }else{
                                        esp = proxB;
                              }
                              try {
                                        Thread.sleep(esp);
                              } catch (InterruptedException e)
                              {
                                        e.printStackTrace();
                              }
                    }
          }
}
```

Miremos una solución concurrente

- La cosa mas simple sería ejecutar dos algoritmos concurrentemente
  - Uno mostrando el mensajes “Hola” cada 3 segundos
  - El otro, mostrando el mensaje “Como estas?” cada 5 segundos

Threads en Java

- La clase `Thread` provee la API para manejar threads y comportamiento genérico
- Un thread debe proveer un método `run()`
  - Run contiene el código que el thread ejecutará cuando sea iniciado

Thread para el mensaje “Hola”

- Crear una clase que extiende Thread (herencia)
  - `public class HolaThread extends Thread`
- Dar una implementación para la operación
  - `public void run(){…}`
- Para activar el thread, se debe crear una instancia de la clase e invocar la operación `run()`
  - `new HolaThread().start();`

Solución

```
public class HolaThread extends Thread{
        public void run()
        {
                while (true){
                        System.out.println("Hola");
                        try {
                                Thread.sleep(3000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
        public static void main(String args[])
        {
                 new HolaThread().start();
        }
}
```

Alternativa

- Definir a la clase `HolaThread` como implementando a la interface `Runnable`
  - `public class HolaThread implements Runnable`
- Dar la implementación para la operación `run`
- Para activar un thread:
  - Crear una instancia de `Thread` utilizando como parámetro una instancia de la clase `HolaThread`
  - Ejecutar el mensaje `start()`
  - `new Thread(new HolaThreadRunnable()).start();`

Solución

```
public class HolaThreadRunnable implements Runnable{
        public void run()
        {
                while (true){
                        System.out.println("Hola");
                        try {
                                Thread.sleep(3000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
        public static void main(String args[])
        {
                 new Thread(new HolaThreadRunnable()).start();
        }
}
```

Versión utilizando una clase anónima

```
public class HolaThreadAnonimo {
        public static void main(String args[])
        {
                    Thread hola = new Thread(){public void run()
                        {
                                while (true){
                                        System.out.println("Hola");
                                        try {
                                                Thread.sleep(3000);
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                    };
                    hola.start();
        }
}
```

Solución para los dos Threads

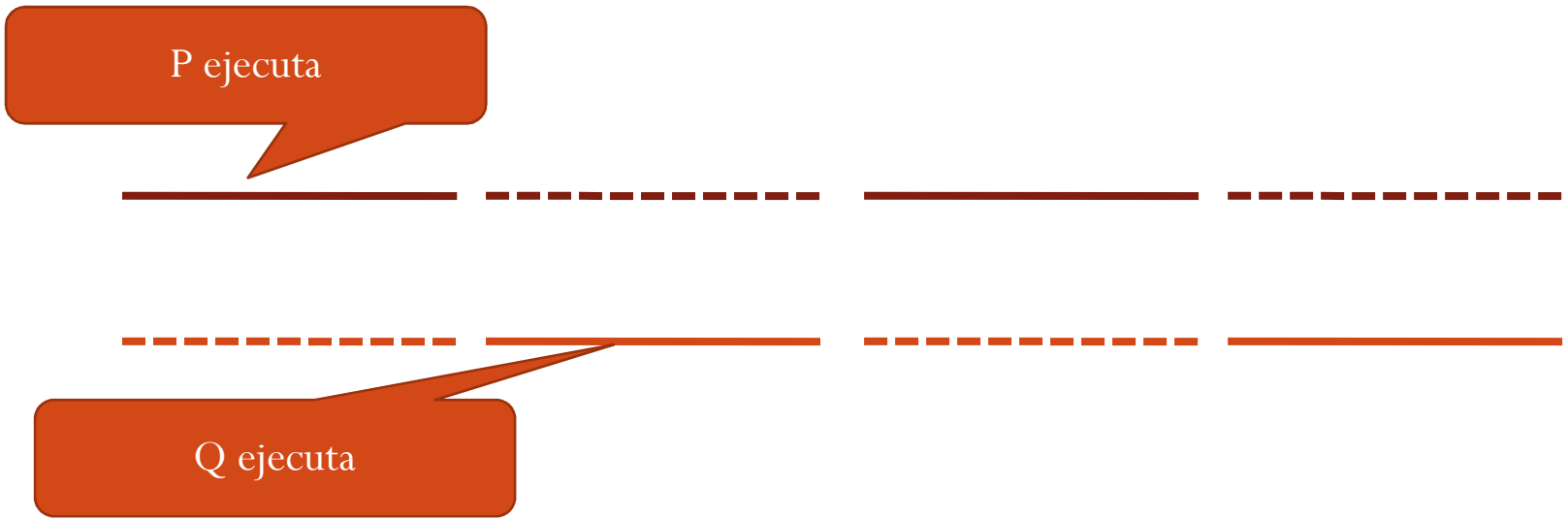
```
public class Mostrar extends Thread {
        int espera;
        String mensaje;

        public Mostrar(int esp, String msg){
                espera=esp;
                mensaje = msg;
        }

        public void run(){
                        while (true){
                                System.out.println(mensaje);
                                try {
                                        Thread.sleep(espera);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                }
        public static void main(String args[])
        {
                new Mostrar(1000, "Hola").start();
                new Mostrar(3000, "Como estas?").start();
        }
}
```

Scheduling de procesos

- En una máquina de von Neumann los threads aparentan ser ejecutados al mismo tiempo pero en realidad su ejecución es intercalada (interleaving)



Scheduling

- Tarea de alternar la ejecución de los threads
- Es responsabilidad del *scheduler*
  - Parte del sistema run-time
  - Ejecutado usando los procesos y el *scheduler* del sistema operativo
- Existen muchos métodos de scheduling

Scheduling

- Scheduling cooperativo: Un thread ejecuta hasta que está en grado de liberar el procesador (terminó, sleep, ejecuta operaciones de I/O)
- Scheduling *pre-emptive*
  - Se interrumple la ejecución de un thread para dar lugar a la ejecución de otro thread (e.g. time-slicing)

Tipos de comportamiento de procesos

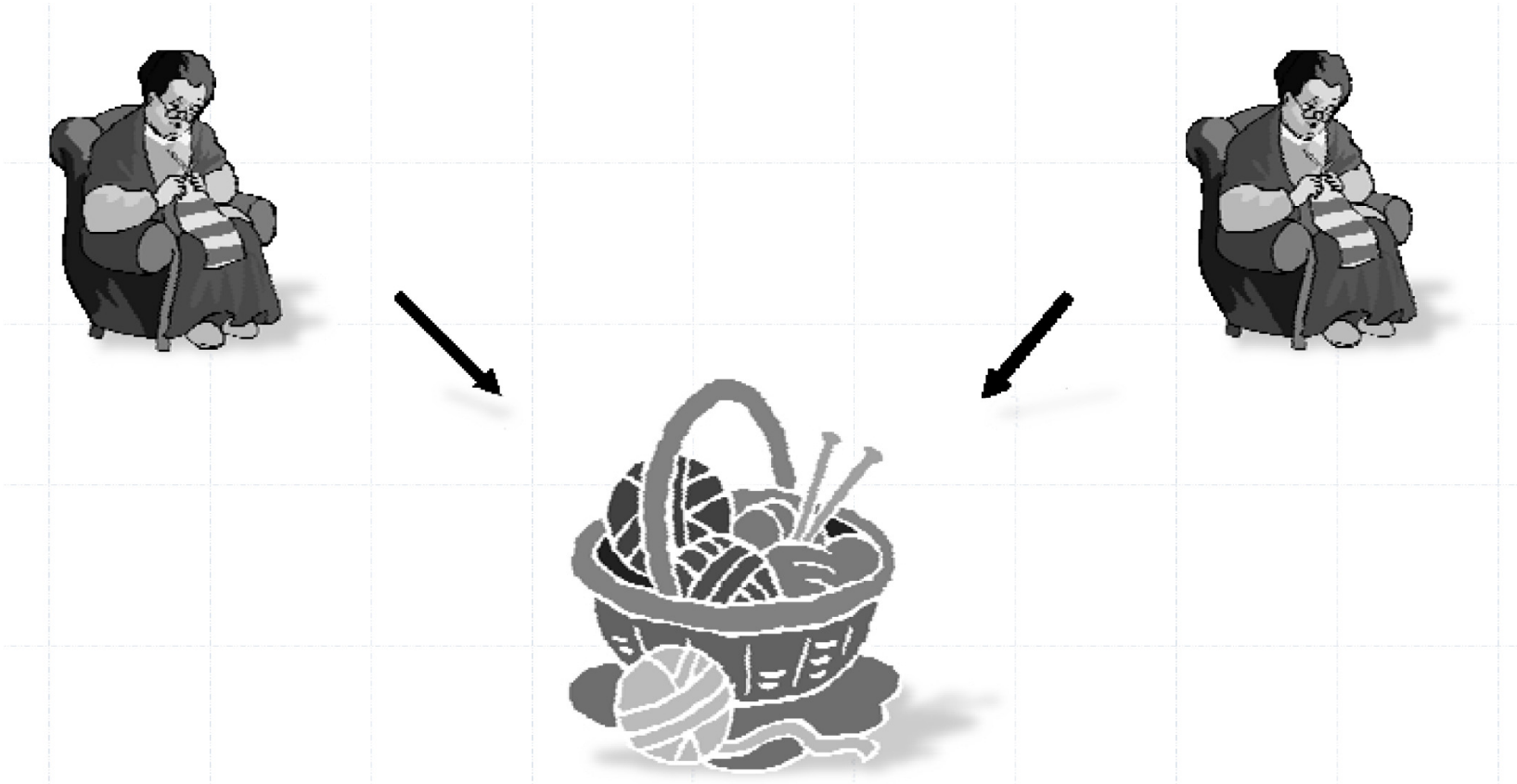
- Procesos independientes
  - Poco interesantes
- Competitivo
  - Comunes en los sistemas operativos y de redes
  - Comparten recursos
- Cooperativos
  - Los procesos se combinan para resolver una tarea en común

Comportamiento de procesos

- Diseñar sistemas concurrentes tiene que ver principalmente con la sincronización y la comunicación entre procesos

Procesos independientes

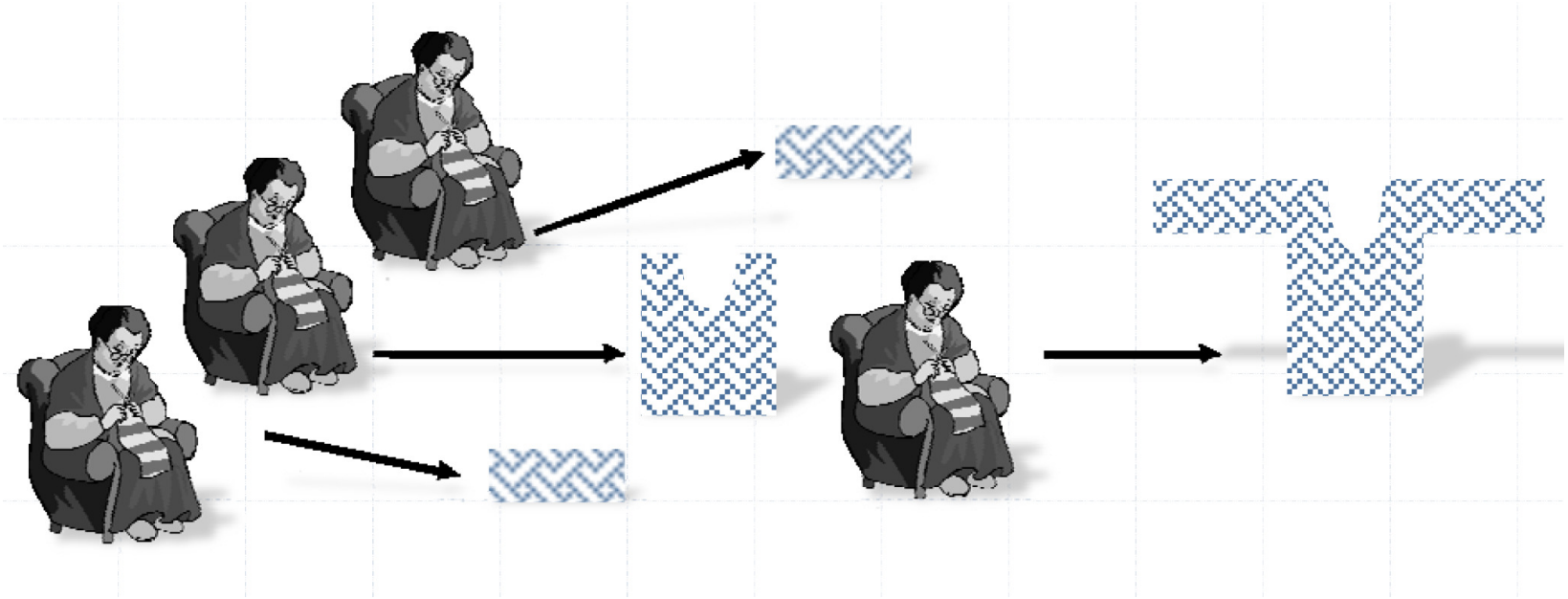
Compitiendo



Compitiendo

Compitiendo

Cooperación



Atomicidad

- Que sucedería si “println” es no atómico

Ejemplo: dos entradas al subte

- Suponer que hay dos molinetes de ingreso
- En cada uno entran personas
  - Las personas llegan con una separación aleatoria (un numero *random* entre 0 y 100 milisegundos)
- Escribir un programa donde el ciclo de ingreso de personas en cada molinete corre en un thread . Existe una variable compartida que cuenta el total de personas ingresadas
- Para generar números aleatorios
  - `Random aleatorio = new Random();`
  - `aleatorio.nextInt(100)`

Solución

```
import java.util.Random;
public class Molinete extends Thread {
          public  static int contador= 0 ;
          int id;
          public Molinete(int esp){
            id = esp;
          }

          public void run(){
                    Random aleatorio = new Random();
                              for(int i = 0; i < 100; i++){
                                        int nuevo=
                                                  ++contador;
                                                  contador = nuevo;
                                        System.out.println(id+"- Entra:  "+i );
                                        try {
                                                  Thread.sleep(aleatorio.nextInt(100));
                                        } catch (InterruptedException e) {
                                                  e.printStackTrace();
                                        }
                              }
                              System.out.println(id+"- En total entraron:  "+contador );
                    }
          public static void main(String args[])
          {
                     new Molinete(1).start();
                     new Molinete(2).start();
          }
}
```

Estados y trazas

- Un programa ejecuta una secuencia de acciones atómicas
- Un estado es el valor de las variables del programa en un momento dado
- Una traza (o historia) es una secuencia de estados que pueden ser producidos pro un conjunto de acciones atómicas de un programa

Una mala traza para el caso del molinete

Propiedades de un programa

- Una propiedad es una fórmula lógica que es verdadera para toda traza posible de ejecución.
- Dos tipos comunes de propiedades sobre la correctitud de programas concurrentes
  - Propiedades de *safety*
    - Una traza nunca alcanza un estado "malo"
  - Propiedades de *liveness*
    - Tarde o temprano, toda traza alcanza un estado "bueno"

Sincronización

- Mecanismo para restringir las posibles ejecuciones (trazas) de un programa concurrente
  - El objetivo es garantizar ciertas propiedades de *safety*

Secciones críticas

- Las trazas indeseadas son consecuencia de interleavings en la ejecución del código que implementa “contador++”
- Para resolver este problema debemos garantizar que esta porción de código se ejecuta atómicamente
  - Sin interleavings (ni actividades paralelas)
- Sección crítica es la parte del programa que debe ser ejecutada atómicamete

Exclusión mutua (Mutex)

- Exclusión mutua: garantiza que sólo un proceso se encuentra ejecutando su sección crítica
- Cómo se obtiene:
  - En teoría bastan variables compartidas
  - En la práctica:
    - los lenguajes de programación proveen mecanismos (semáforos, monitores, …)
    - Comunicación sin variables compartidas
    - Soporte de hardware (instrucciones atómicas especiales)

Problema de la exclusión mutua

Thread

Sección no crítica

Entrada a la sección crítica

Sección crítica

Salida de la sección crítica

Sección no crítica

Problema de la exclusión mutua

- Supuestos
  - No se comparten variables entre la sección crítica y la sección no crítica (ni con el protocolo de ingreso/egreso de la sección crítica)
  - La sección crítica siempre termina
  - Lectura/Escritura de variables son atómicas
  - El *scheduler* es debilmente *fair*
    - *Un proceso que espera para ejecutar , en algún momento podrá ejecutar*

Problema de la exclusión mutua

- Requerimiento 1: Mutex
  - En cualquier momento, al máximo un proceso está en su región crítica
- Requerimiento 2: ausencia de *deadlocks* y de livelocks
  - Si varios procesos intentan entrar a su sección crítica, uno lo logrará
- Requerimiento 3: Garantía de entrada
  - Un proceso intentando entrar a su región crítica lo logrará tarde o temprano

Mutex: Algoritmo I

- Usar una variable compartida turno que indica quien puede entrar a la sección
- Consideramos turno inicializada en 0

ThreadId = 0

```
Sección no crítica
while (turno != ThreadId) {}
Sección crítica
Turno = (Threadid +1 % 2)
Sección no crítica
```

ThreadId = 1

```
Sección no crítica
while (turno != ThreadId) {}
Sección crítica
Turno = (Threadid +1 % 2)
Sección no crítica
```

Mutex: Algoritmo I

- Análisis
  - Mutex: Si
  - Ausencia deadlocks/livelocks: Si
  - Garantía de entrada: NO
    - Pensar que sucede si no termina la sección no crítica

Mutex: Algoritmo II

- Usar una variable indicando flag quién entro a la sección crítica:
  - Valor inicial {false,false}:

ThreadId = 0

```
Sección no crítica
while (flag[ThreadId +1 % 2]){}
Flag[Threadid] = true;
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

ThreadId = 1

```
Sección no crítica
while (flag[ThreadId +1 % 2]){}
Flag[Threadid] = true;
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

Mutex: Algoritmo II

- Análisis
  - Mutex: No
  - Ausencia deadlocks/livelocks: Si
  - Garantía de entrada: Si

Mutex: Algoritmo III

- Usar flag para indicar quien quiere entrar:
  - Valor inicial {false,false}:

ThreadId = 0

```
Sección no crítica
Flag[Threadid] = true;
while (flag[ThreadId +1 % 2]){}
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

ThreadId = 1

```
Sección no crítica
Flag[Threadid] = true;
while (flag[ThreadId +1 % 2]){}
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

Mutex: Algoritmo III

- Usar flag para indicar quien quiere entrar:
  - Valor inicial {false,false}:

ThreadId = 0

```
Sección no crítica
Flag[Threadid] = true;
while (flag[ThreadId +1 % 2]){}
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

ThreadId = 1

```
Sección no crítica
Flag[Threadid] = true;
while (flag[ThreadId +1 % 2]){}
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

Mutex: Algoritmo de Dekker(I + III)

ThreadId = 0

```
otro = ThreadId +1 % 2
Sección no crítica
flag[ThreadId] = true;
while(flag[otro]){
    if turno = otro then
        flag[ThreadId] = falso;
        while(!turno = ThreadID) {}
         flag[ThreadId] = true;
}
Sección crítica
Turno = otro;
Flag[Threadid] = false;
Sección no crítica
```

ThreadId = 1

Mutex: Algoritmo de Peterson (I + III)

ThreadId = 0

```
otro = ThreadId +1 % 2
Sección no crítica
flag[ThreadId] = true;
Turno = otro;
while (flag[otro] && turno == otro){}
Sección crítica
Flag[Threadid] = false;
Sección no crítica
```

ThreadId = 1

Dekker y Peterson

- Análisis
  - Mutex: Si
  - Ausencia deadlocks/livelocks: Si
  - Garantía de entrada: Si

- Prueba formal de estas propiedades es bastante compleja (no la veremos en el curso)

Algoritmo Bakery

- Entrando: array [1.. N] of bool = {false}
- Numero: array [1..N] of integer = {0}

```
Sección no crítica
Entrando[ThreadId] = true;
Numero[ThreadId] = 1+max(Numero[1],…, Numero[N]);
Entrando[ThreadId] = false;
For (j =1, j <= N, j++) {
      while (Entrando[j] ){}
      while((Numero(j)!=0 && ((Numero[j],j) < (Numero[ThreadId], ThreadId))
{}
Sección crítica
Numero[[Threadid] = 0;
Sección no crítica
```