Asignatura 780014
Programación Avanzada

Tema 4 –MECANISMOS PARA LA EXCLUSIÓN MUTUA

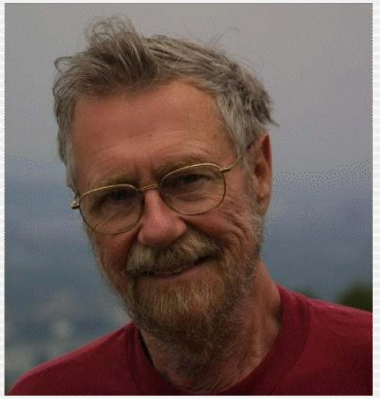
Mecanismos para la exclusión mutua

- Objetivo del tema:
  - Completar la revisión de mecanismos de exclusión mutua existentes y del problema del interbloqueo

Índice

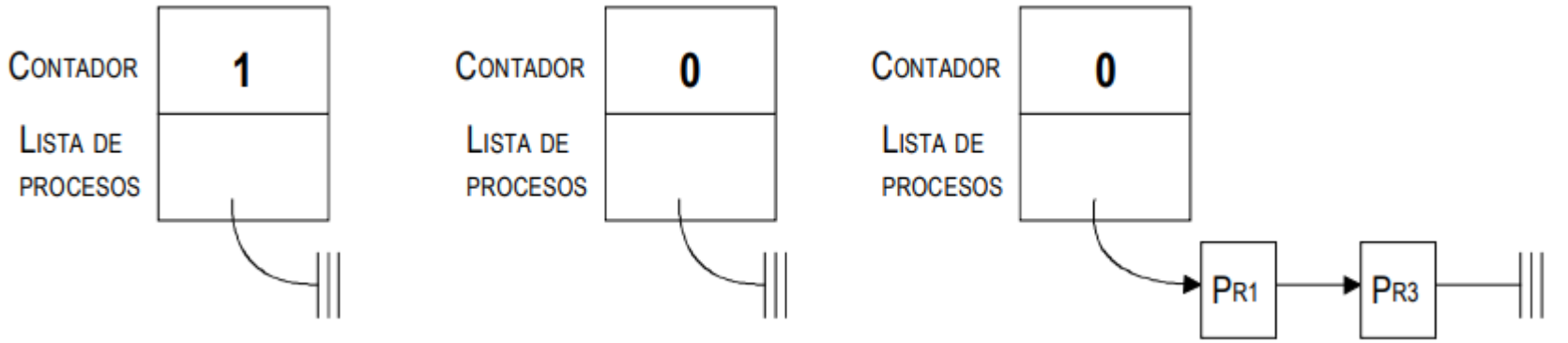
Semáforos

- Tipo Abstracto de Datos (TAD), concebido por Dijkstra, que resuelve problemas de concurrencia



Edsger W. Dijkstra

- Estructura
  - Contador
  - Cola de procesos (hilos)



Semáforos

- Definición inicial (operaciones):
  - **Declaración**: VAR s: SEMAPHORE;
  - **Initial(s, v)**: Inicializa el contador de s
  - **Wait(s)** (definida como P por Dijkstra): Si el contador es mayor que cero, lo decrementa en uno. Sino, suspende al proceso en la cola.

```
PROCEDURE wait(s:SEMAPHORE);
  BEGIN (* Operación Atómica *)
      IF (s.contador>0) THEN
            s.contador := s.contador - 1;
      ELSE
            suspende_en(s.cola);
   END;
```

  - **Signal(s)** (definida como V por Dijkstra): Si la cola de procesos está vacía, incrementa en uno el contador. Si existen procesos en cola, activa uno.

```
PROCEDURE signal(s:SEMAPHORE);
    BEGIN (* Operación Atómica *)
        IF vacia(s.cola) THEN
            s.contador := s.contador + 1;
        ELSE
            activar_desde(s.cola);
    END;
```

Semáforos

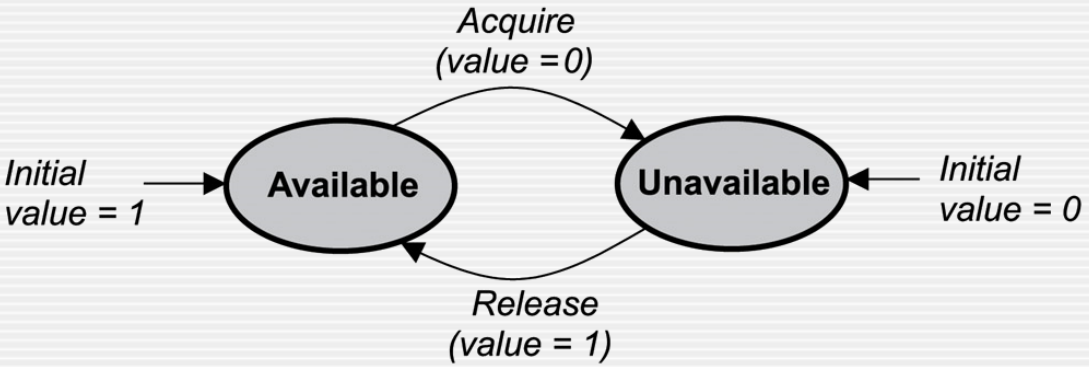
- Atomicidad
  - **wait** y **signal** se realizan de forma **atómica**
- Invariantes
  - a) S >= 0
  - b) $S = S_0 + \#signal - \#wait$

S = valor del contador
#signal = cantidad de signals ejecutados en S
#wait = cantidad de waits completados en S

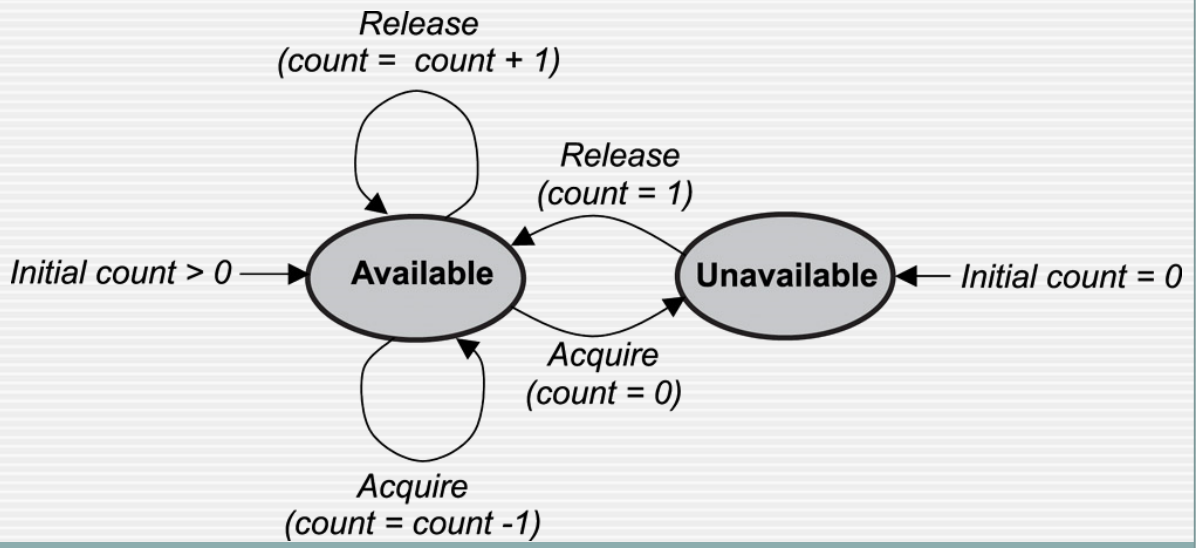
- Usos de los semáforos
  - Comunicación
  - Sincronización
  - Comunicación sincronizada

Semáforos

- Tipos de semáforos
  - **Binarios**: 0 ó 1



  - **De contador**: 0 .. N



Semáforos en Java: clase Semaphore

- public class Semaphore
  - Un semáforo contiene un número de **permisos** (*permits*)=contador
    - **Binarios**: permisos = **1**
    - **Con contador**: permisos = **N**
  - Un semáforo puede crearse como '**justo**' (*fair*)
    - Si y sólo si se usa el constructor con parámetro **boolean** y valor **true**
    - Libera los hilos bloqueados en el orden de llegada (FIFO)
  - Constructores
    - `Semaphore(int permits)`
    - `Semaphore(int permits, boolean fair)`

Semáforos en Java: clase Semaphore

- Operaciones más importantes
  - `void` **`acquire()`**:
    - Si hay permisos disponibles, adquiere uno y resta 1 al nº de permisos disponibles. El hilo puede proseguir
    - Si no hay permisos disponibles, el hilo actual se bloquea hasta que alguien invoque a *release* (y el hilo sea seleccionado para ejecutarse) o hasta que alguien interrumpe el hilo
  - `void` **`acquire(int n)`**: ídem, pero con varios permisos de una vez
  - `void` **`release()`**: añade 1 permiso a los disponibles, liberando potencialmente a un proceso bloqueado en *acquire()*
  - `void` **`release(int n)`**: ídem, pero con varios permisos de una vez
  - `boolean` **`tryAcquire()`**: comprueba si el semáforo tiene permisos (útil para que lo ejecute un hilo antes de bloquearse)
- Más información en la documentación oficial de la clase Semaphore

Semáforos en Java: clase Semaphore

- Se utilizan para controlar el número de hilos que pueden acceder a un recurso (comunicación)
  - Un semáforo **inicializado a 1** se comporta como un semáforo **binario**
    - Para proteger zonas en exclusión mutua
- Para simular condiciones de espera (sincronización)
  - Un semáforo **inicializado a 0** "simula" un await de un Condition
- Diferencia semáforo/lock:
  - Un hilo bloqueado en un **semáforo** puede ser **liberado por otro**
  - Un **lock** sólo lo puede **liberar el** hilo **que lo tiene adquirido**
  - Un **semáforo** puede permitir pasar a **más de un hilo** a la vez, el **cerrojo sólo** a **uno**

Semáforos en Java: clase Semaphore

- Ejemplo de secuencia (visto en temas 2 y 3) con semáforos:

```
public class Secuencia {
    private int valor=0;
    private Semaphore sem = new Semaphore(1);
    public int getSiguiente() throws
    InterruptedException {
        try {
            sem.acquire();
            valor++;
            return valor;
        } finally {
            sem.release();
        }
} }
```

```
public class Main {
    public static void main(String[] args) {
        Secuencia sec=new Secuencia();
        for(int i=0; i<100; i++) {
            Hilo hilo=new Hilo(i, sec);
            hilo.start();
        }
    }
}
```

```
public class Hilo extends Thread {
    private Secuencia sec;
    private int id;
    Hilo(int id, Secuencia sec) {
        this.id=id;     this.sec=sec;
    }
    public void run() {
        System.out.println("Hilo "+id+": secuencia="+sec.getSiguiente());
    }
}
```

```
run:
Hilo 2: secuencia=1
Hilo 3: secuencia=3
…
Hilo 98: secuencia=99
Hilo 95: secuencia=100
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo de Semáforos

- Implementación del productor-consumidor con **semáforos**

```
public class Main
{
    public static void main(String[] args)
    {
        Buffer buf = new Buffer(10);
        Thread prod1 = new Productor(buf, "Prod1");
        Thread prod2 = new Productor(buf, "Prod2");
        Thread cons = new Consumidor(buf, "Cons");
        prod1.start();
        prod2.start();
        cons.start();
    }
}
```

Ejemplo de Semáforos

```
public class Productor extends Thread
{
    Buffer buf;
    String id;
    public Productor(Buffer buf, String id)
    {
        this.buf = buf;
        this.id = id;
    }
    public void run()
    {
        Object msg;
        for (int i = 1; i <= 20; i++)
        {
            try {
                sleep(100 + (int) (200 * Math.random()));
                msg = (Object) (id + " - " + i);
                buf.insertar(msg);
                System.out.println("Produzco: "+msg);
            } catch (InterruptedException e) {}
        }
    }
}
```

```
public class Consumidor extends Thread
{
    Buffer buf;
    String id;
    public Consumidor(Buffer buf, String id)
    {
        this.buf = buf;
        this.id = id;
    }

    public void run()
    {
        Object msg;
        for (int i = 1; i <= 20; i++)
        {
            try {
                sleep(100 + (int) (200 * Math.random()));
                msg=buf.extraer();
                System.out.println("Consumo: "+msg);
            } catch (InterruptedException e) {}
        }
    }
}
```

Ejemplo de Semáforos

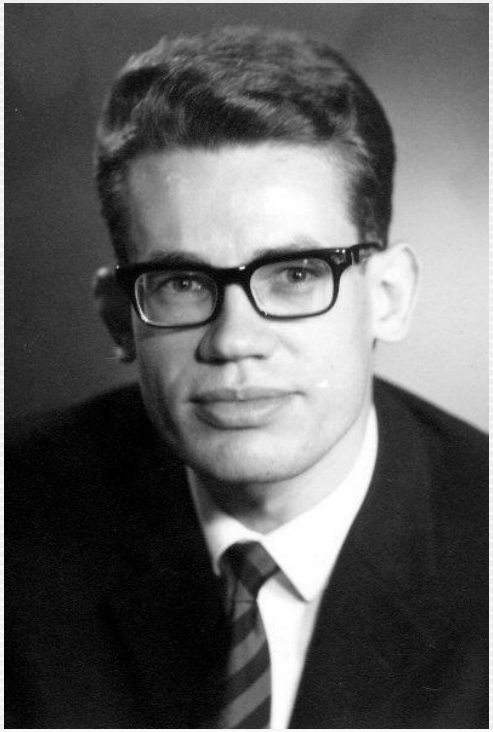
```
public class Buffer {
    private Object[] buf;
    private int in=0, out=0, numElem=0, maximo=0;
    private Semaphore vacio = new Semaphore(0);
    private Semaphore lleno;
    private Semaphore em = new Semaphore(1); //Exclusión mutua

    public Buffer(int max)
    {
        this.maximo = max;
        buf = new Object[max];
        lleno = new Semaphore(max);
    }
    public void insertar(Object obj) throws InterruptedException
    {
        lleno.acquire();
        em.acquire(); //Bloqueo: Comienza SC
        buf[in] = obj;
        numElem++;
        in = (in + 1) % maximo;
        em.release(); //Desbloqueo: Finaliza SC
        vacio.release();
    }
    public Object extraer() throws InterruptedException
    {
        vacio.acquire();
        em.acquire(); //Bloqueo: Comienza SC
        Object obj;
        obj = buf[out];
        buf[out] = null;
        numElem = numElem - 1;
        out = (out + 1) % maximo;
        em.release(); //Desbloqueo: Finaliza SC
        lleno.release();
        return obj;
    }
}
```

Regiones Críticas

- *Per Brinch Hansen* propuso el concepto de "**Región Crítica**" (RC) asociada a una **variable compartida**, de manera que un proceso que entrase dentro de dicha región crítica tenía garantizada la **exclusión mutua** para acceder a ella

En Concurrent Pascal:

```
...
VAR v: SHARED INTEGER;
.....
PROCEDURE P1;
BEGIN
  ...
  REGION v DO v:= v + 2;
  .....
END;
.....
```



Regiones Críticas

- La semántica de la RC establece que:

1. Los procesos (hilos) concurrentes sólo pueden **acceder** a las variables compartidas **dentro de sus** correspondientes **RC**
2. Un proceso que quiera **entrar** a una RC lo hará en un **tiempo finito**
3. En un instante *t* de tiempo, sólo **un proceso** puede estar dentro de **una RC de una variable** compartida determinada
   - Esto no impide que, por ejemplo, un proceso P1 esté dentro de la RC asociada con la variable *v* y un proceso P2 dentro de la RC asociada con la variable *w*. Es decir, las **RC** que hacen referencia a **variables distintas** pueden ejecutarse **concurrentemente**
4. Un proceso **está dentro** de una RC **un tiempo finito**, al cabo del cual la abandona

Regiones Críticas

- Consecuencias:
  1. Si el número de procesos dentro de una RC es igual a **0**, un proceso que lo desee puede **entrar** a dicha RC
  2. Si el número de procesos dentro de una RC es igual a **1**, y N procesos quieren entrar, esos N procesos deben **esperar**
  3. Cuando un **proceso sale** de una RC, **se permite que entre uno** de los procesos que esperan
  4. Las decisiones de quién entra y cuándo se abandona una RC se toman en un **tiempo finito**
  5. La puesta en cola de **espera** puede ser **justa o no**
  6. La **espera** que realizan los procesos es **pasiva** (si un proceso intenta acceder a una RC y está ocupada, abandona el procesador en favor de otro proceso)

Regiones Críticas en Java

- La cláusula **synchronized** siempre está referida a un **objeto**
  - Todos los métodos/bloques **synchronized** definidos en un objeto están dentro de una única región crítica
- Un objeto implementa una **RC** estricta si:
  - Todas sus **variables** son **private**
  - Y todos sus **métodos** son **synchronized**

Regiones Críticas en Java

- ¿Cerrojo implícito o Región Crítica?

```
public class VarComp {
  private int v=0;
  public void incrementar(int i) {
     v=v+i;
  }

public class Proceso extends Thread {
  VarComp var;
  …
  public void run()
  {
     …
     synchronized (var)
     {
         var.incrementar(12);
         …
     }
 …
```

```
public class VarComp {
  private int v=0;
  public synchronized void incrementar(int i) {
     v=v+i;
  }

public class Proceso extends Thread {
  VarComp var;
  …
  public void run()
  {
     …
     var.incrementar(12);
     …
 …
```

Regiones Críticas Condicionales

- Las RC si se usan para sincronización = Espera activa
  - Para **evitar espera activa** necesitamos **bloquear un proceso** hasta que se cumpla la condición

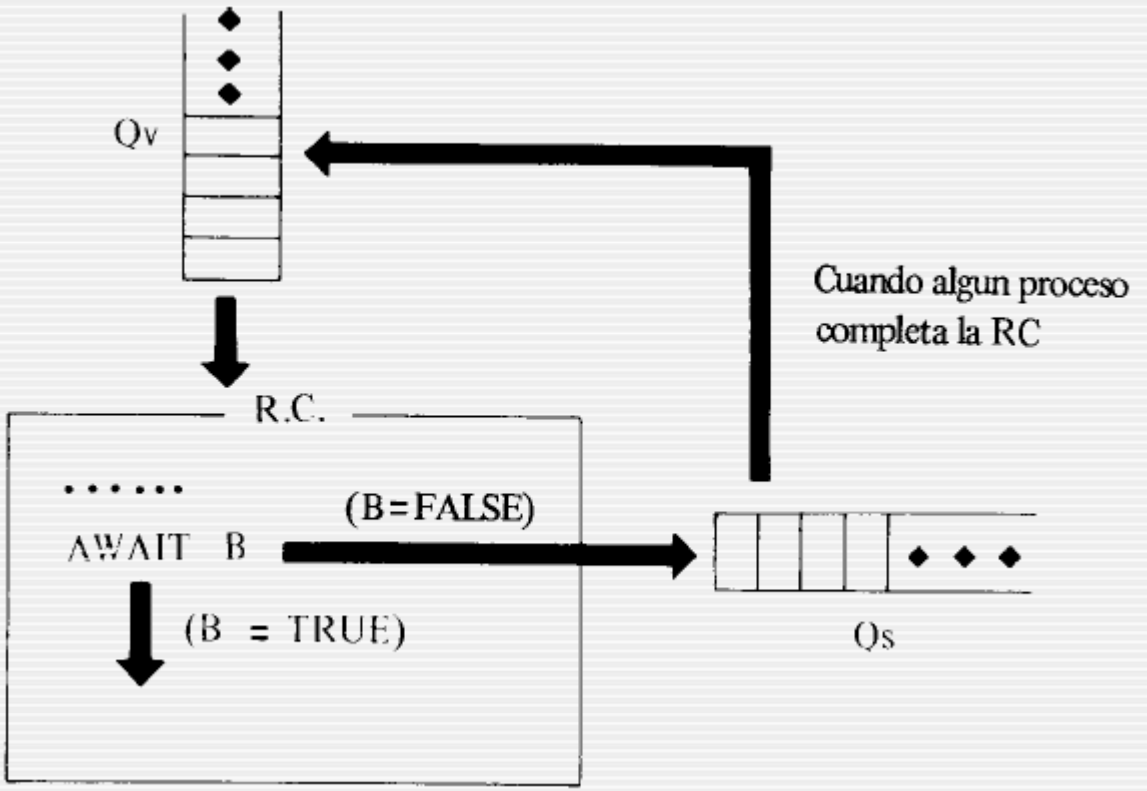
- Una Región Crítica Condicional (RCC) permite a un proceso **abandonar temporalmente la RC**, si una condición no se cumple, a la espera de que otro proceso modifique dicha condición

Regiones Críticas Condicionales

- Las RCC permiten abandonar temporalmente la RC
  - Si una condición no se cumple
  - A la espera de que otro proceso entre en la RC y salga
  - La condición puede haber sido modificada
  - El proceso que salió temporalmente, vuelve a entrar a comprobar la condición
- No hay espera activa

Regiones Críticas Condicionales

- La sentencia AWAIT hace que el proceso dentro de la RCC, la abandone temporalmente
- Cuando un proceso completa la RCC, saca a **todos** los que esperan
- En Java no tiene implementación directa

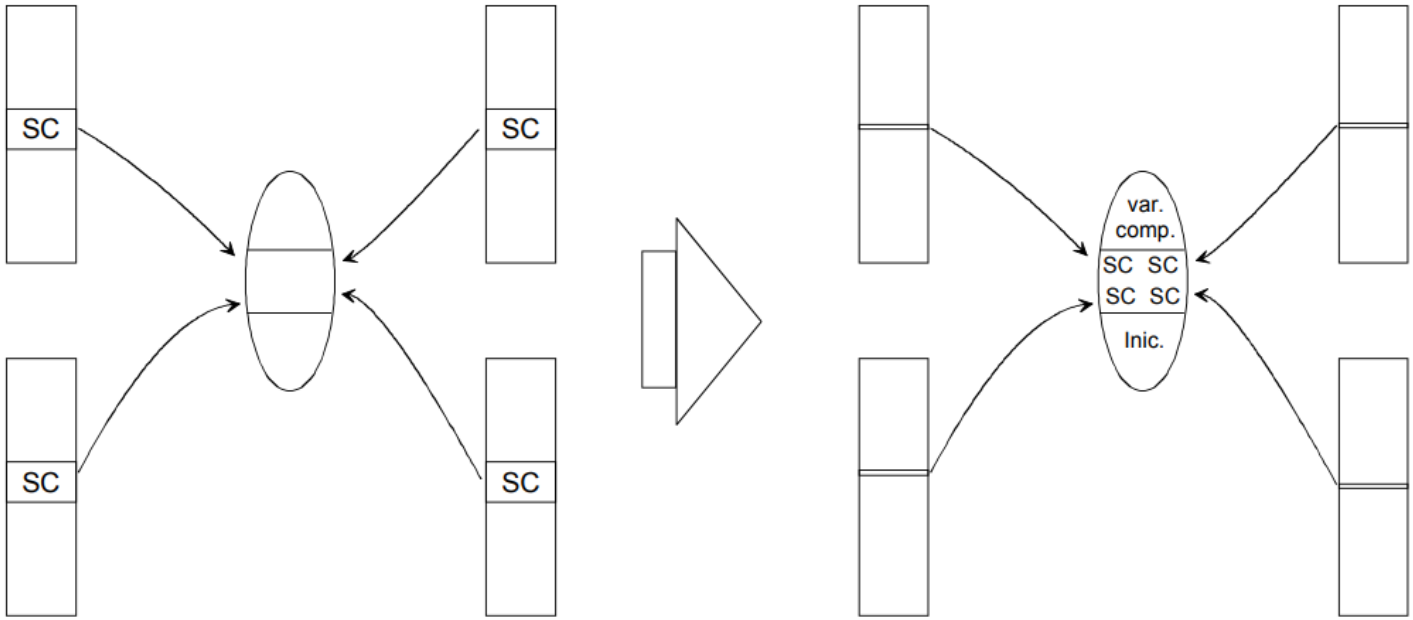


Monitores

Sir Charles Antony Richard Hoare
(Tony Hoare)

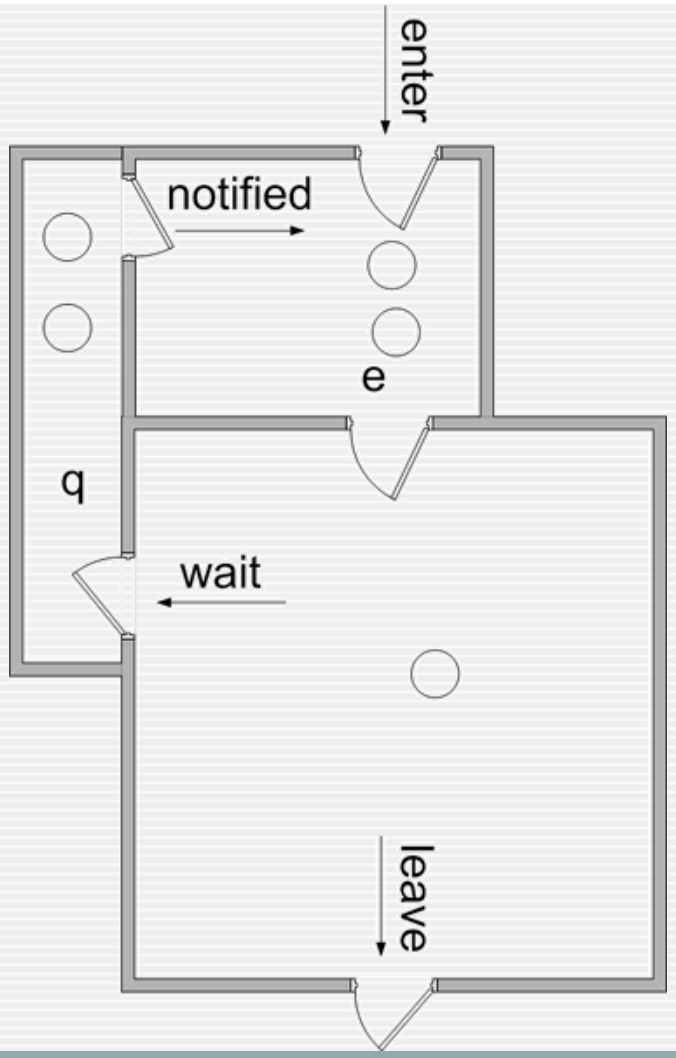
- Un monitor implementa una RCC especial:
  - Un hilo **dentro** del monitor puede sacar de la cola de espera **a uno o a todos** los hilos esperando
  - Engloba **datos compartidos**, los **métodos** que los usan y una sección de inicialización que garantiza que el monitor parte de un estado coherente

Edsger W. Dijkstra



Monitores

- Un proceso que llega al monitor tiene que esperar a que no haya nadie dentro
- Comprueba si se satisface una condición
  - Si cumple la condición, continúa ejecutando
  - Si no se cumple, tiene que esperar (*wait*) en una cola *q* asociada
- Si modifica la condición, puede notificar a los procesos que esperan en *q*, para que vuelvan a entrar (*notify*)



Monitores de Mesa en Java

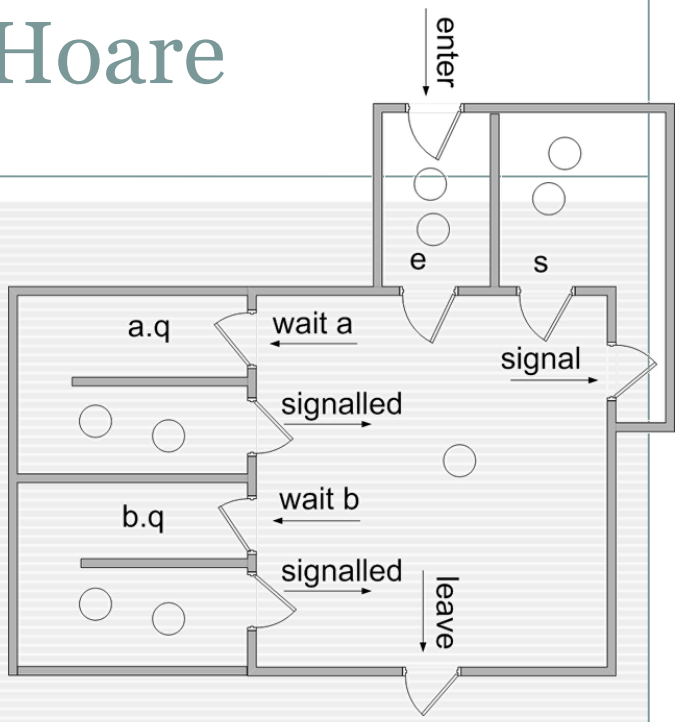
- Java dispone de tres sentencias para implementar un monitor:
  - **wait()**: el proceso que la ejecuta (dentro de una RC) se queda bloqueado hasta que otro proceso lo saque de esa situación
  - **notify()**: ejecutada por un proceso (dentro de una RC), saca a uno de los procesos en *wait* de su estado
  - **notifyAll()**: igual que la anterior, pero saca a todos los procesos en *wait* dentro de esa RC

- Igloo de los ejemplos del tema 3 implementado como monitor (*atención a la forma en que se comprueba la condición*):

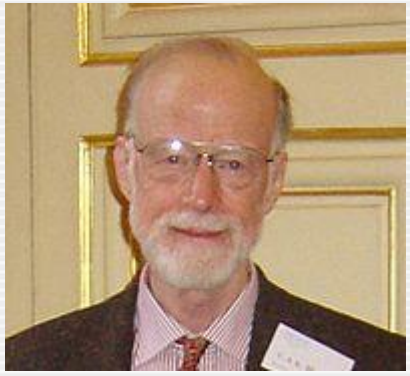
```
public class Igloo // Monitor
{
    private boolean pescando=false;

    public synchronized void pescar()
    {
        while(pescando)
        { //El recurso está ocupado
            try {
                wait(); //Sale temporalmente de la SC
            } catch(InterruptedException e){}
        }
        pescando=true;  //El proceso adquiere el recurso
    }

    public synchronized void noPescar()
    {
        pescando=false;  //El proceso libera el recurso
        notifyAll();     //Libera los procesos que estén
                         //esperando el recurso
    }
}
```

Monitores de estilo Hoare

- Propuesta original de Hoare y Hansen
  - Proponen un monitor con **varias colas**
    - Tantas como se necesiten
    - Cada una implica una condición distinta
    - Todas las colas son justas (*fair*)
- Comportamiento:
  - Thread A espera una condición específica
    - Hace *wait()* de una cola para esa condición
  - Thread B ejecuta un *signal()*
    - Abandona temporalmente el monitor
    - Espera hasta que el thread A despertado salga del monitor o ejecute un nuevo *wait()*

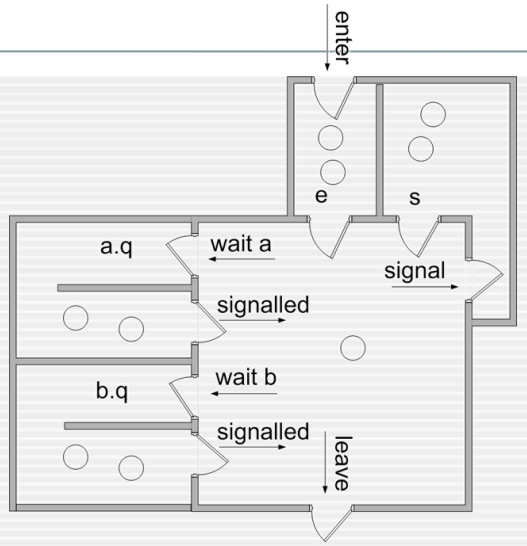


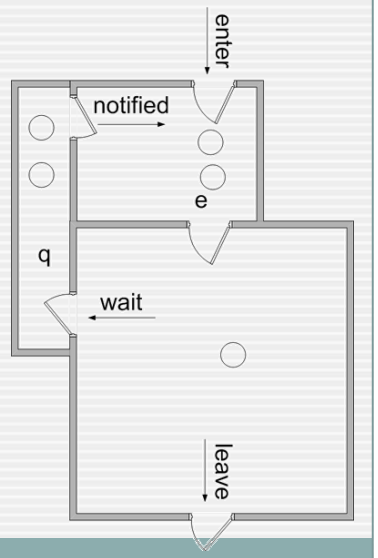
Monitor de estilo Hoare


Sir Charles Antony Richard Hoare (Tony Hoare)

Monitores de Mesa vs Monitores de Hoare

- Un monitor de Hoare, garantiza que:
  - Si un proceso sale de su cola de condición, es porque dicha condición se cumple y no ha dejado de cumplirse
    - No es necesario comprobar la condición
  - El proceso que detecta la condición y despierta a otro, sale del monitor en ese momento
- Un monitor simple (de Mesa), no garantiza:
  - El cumplimiento de la condición de un proceso despertado
    - Es necesario comprobar la condición
  - Porque:
    - El proceso que despierta a otro se sigue ejecutando
    - Otros procesos pueden haber cambiado la condición





Ejemplos

- Implementación del productor-consumidor con **monitores**

```
public class Buffer
{
    private Object[] buf;
    private int in=0, out=0, numElem=0, maximo=0;

    public Buffer(int max)
    {
        this.maximo = max;
        buf = new Object[max];
    }
    public synchronized void insertar(Object obj) throws
InterruptedException
    {
        while (numElem==maximo) //Buffer lleno
        {
            wait();
        }
        buf[in] = obj;
        numElem++;
        in = (in + 1) % maximo;
        notifyAll();
    }
```

```
public synchronized Object extraer() throws
InterruptedException
{
        while (numElem==0) //Buffer vacío
        {
            wait();
        }
        Object obj;
        obj = buf[out];
        buf[out] = null;
        numElem = numElem - 1;
        out = (out + 1) % maximo;
        notifyAll();
        return obj;
    }
}
```

```
run:
Produzco: Prod1 - 1
Consumo: Prod1 - 1
Produzco: Prod2 - 1
Consumo: Prod2 – 1
…
Consumo: Prod2 - 19
Consumo: Prod2 – 20
BUILD SUCCESSFUL (total time: 7 seconds)
```

Ejemplos

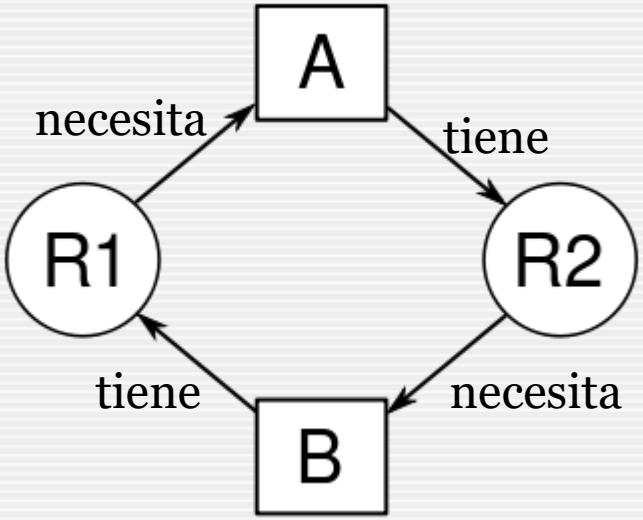
- Implementación del buffer del productor-consumidor con **semáforos** y **locks**

```
public class Buffer {
    private Object[] buf;
    private int in = 0, out = 0, numElem = 0, maximo = 0;
    private Semaphore vacio = new Semaphore(0);
    private Semaphore lleno;
    private Lock control = new ReentrantLock(); //Exclusión mutua
    public Buffer(int max) {
        this.maximo = max;
        buf = new Object[max];
        lleno = new Semaphore(max);
    }
    public void insertar(Object obj) throws InterruptedException {
        lleno.acquire();
        control.lock(); //Bloqueo: Comienza SC
        try {
            buf[in] = obj;
            numElem++;
            in = (in + 1) % maximo;
            vacio.release();
        } finally {
            control.unlock(); //Desbloqueo: Finaliza SC
        }
    }
```

```
public Object extraer() throws InterruptedException {
        vacio.acquire();
        control.lock(); //Bloqueo: Comienza SC
        Object obj;
        try {
            obj = buf[out];
            buf[out] = null;
            numElem = numElem - 1;
            out = (out + 1) % maximo;
            lleno.release();
            return obj;
        } finally {
            control.unlock();//Desbloqueo: Finaliza SC
        }
  }
}
```

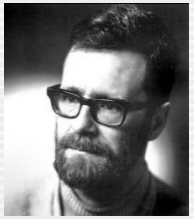
Interbloqueo

- Un interbloqueo (deadlock) impide que un programa progrese
  - Los hilos se quedan bloqueados esperándose unos a otros
- Tienen que darse cuatro condiciones necesarias y suficientes
- Propósito: **evitar deadlocks**
  - Diseñar sistemas en los que no puedan ocurrir interbloqueos



Interbloqueo

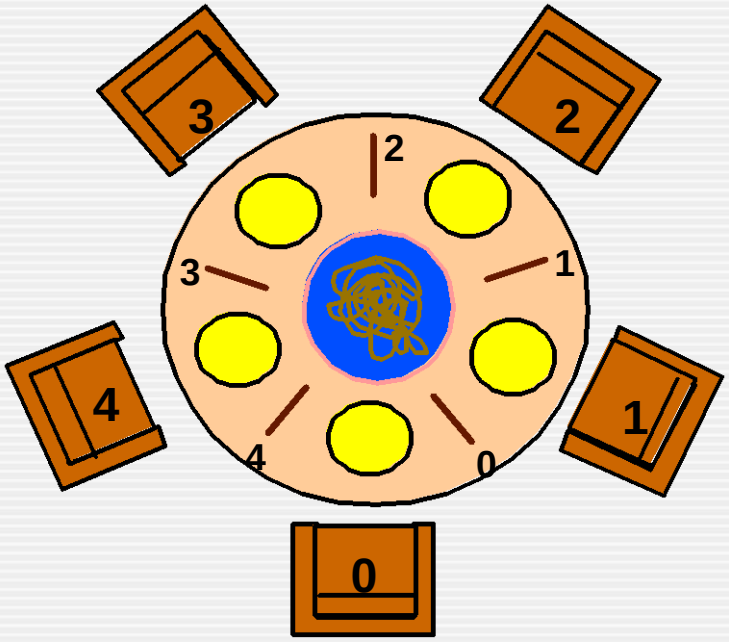
- Ejemplo: los filósofos comensales (Dijkstra)
  - Cinco filósofos están sentados alrededor de una mesa circular
  - Cada uno de ellos alterna entre "**pensar**" y "**comer**"
  - En medio de la mesa hay un gran plato de spaghetti, y cada filósofo **necesita dos palillos** para poder comer
  - Como los filósofos cobran menos que los informáticos, sólo tienen **5 palillos en total**
  - Hay un palillo colocado a la **izquierda** y otro a la **derecha** de cada uno
  - Cuando un filósofo tiene hambre:
    - Toma **primero** su palillo **derecho** y **después** el **izquierdo**
    - Con los dos palillos se puede poner a comer



Edsger W. Dijkstra
(1930-2002)

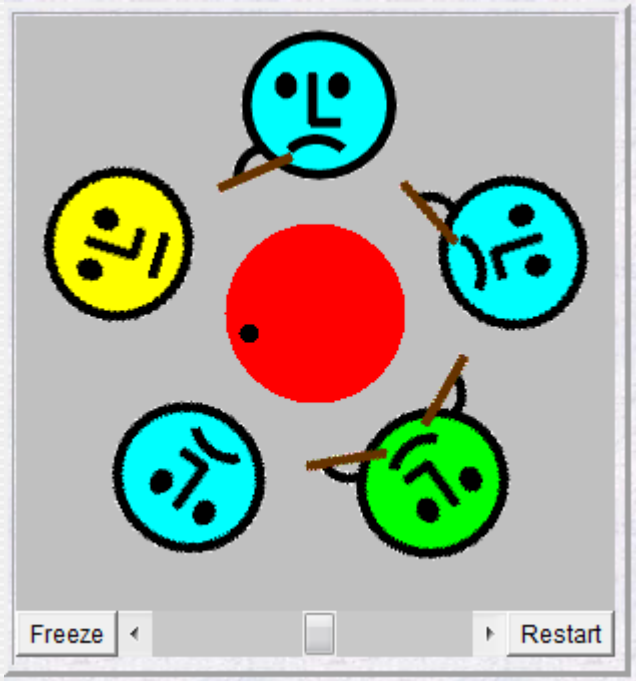
Interbloqueo

- Modelado del problema:
  - Cada palillo es un **recurso compartido**
    - Dos métodos, get() y put(), accedidos en **exclusión mutua**
  - Cada filósofo es un **hilo**
    - En su método run():
      - Simula la acción de pensar
      - Coge el palillo derecho
      - Coge el palillo izquierdo
      - Simula la acción de comer
      - Suelta el palillo derecho
      - Suelta el palillo izquierdo



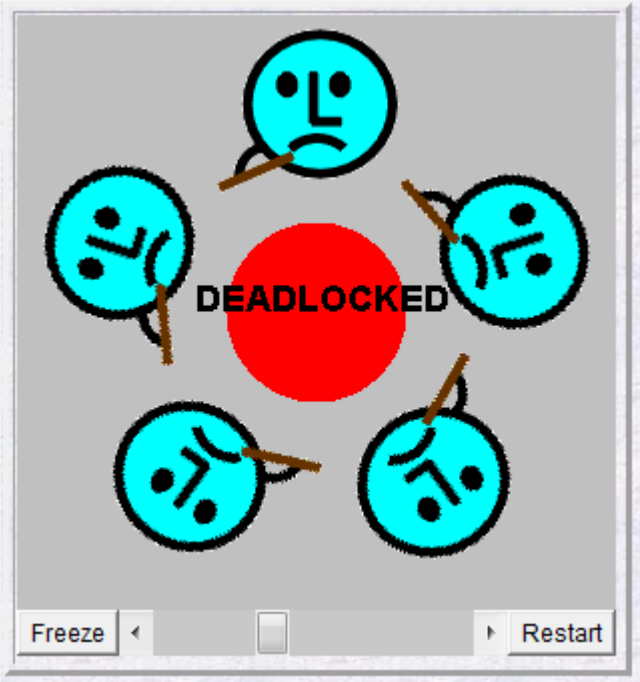
Interbloqueo

- Tal y como está planteado el problema, al cabo de un tiempo más o menos largo, ocurrirá que cada filósofo tiene un palillo en su mano derecha y ninguno puede continuar: Los procesos están **interbloqueados**



Al cabo de un tiempo...

Ejecutar Applet



Interbloqueo

- 4 condiciones (de Coffman) para el interbloqueo

Edward G. "Ed" Coffman, Jr

1. **Exclusión mutua**
   - Al menos existe un recurso compartido, al cual sólo puede acceder un proceso a la vez
2. **Asignación parcial** de recursos
   - Cada proceso va adquiriendo (y bloqueando) los recursos a medida que los va necesitando
3. **No expulsión** de recursos
   - Una vez adquirido un recurso, no se libera hasta no haber adquirido el resto
4. **Espera circular**
   - Cada proceso espera la liberación de un recurso por otro proceso, que a su vez espera un tercero, y así sucesivamente hasta completar el círculo con el primer proceso

Interbloqueo

- Evitar interbloqueo = evitar **alguna** de las condiciones de Coffman:
  - Eliminando la exclusión mutua
    - **Ningún proceso** puede tener **acceso exclusivo** a un recurso
  - Eliminando la asignación parcial de recursos
    - Haciendo que todos los procesos **pidan todos los recursos** que van a necesitar **antes** de empezar
  - Eliminando la condición de no expulsión de recursos
    - **Liberando los recursos** adquiridos **o expropiándoselos** a los hilos que los tengan bloqueados y estén a la espera
  - Eliminando la condición de espera circular
    - Se le permite a un proceso **poseer sólo un recurso** en un determinado momento, **o establecer una jerarquía** para evitar ciclos

El problema de los filósofos en Youtube

- En youtube se puede ver un vídeo en el que se muestran estas soluciones:

- También en tono de humor, otro vídeo:

Ejercicios

- 1.- Ejecutar los códigos utilizados en esta presentación.
- 2.- Crear un programa en Java que simule el problema del sensor de temperatura utilizando un objeto compartido (buffer de tamaño 10) donde se vayan depositando las temperaturas (ejercicio 3 del tema 3), pero utilizando sólo Semáforos.
- 3.- Repetir el ejercicio 2, pero utilizando únicamente Monitores.