# TEMA 2

# Primeras aproximaciones a la solución de los problemas de la Programación Concurrente


# Primeras aproximaciones a la solución de los problemas de la Programación Concurrente

1. Introducción
2. Tipos de sincronización y su solución
3. Soluciones de Espera Ocupada
4. La realidad del Hardware
5. Soluciones Hardware

# 1. Introducción

- Los procesos concurrentes necesitan reglas de sincronización para controlar sus relaciones.
- Existen **recursos no compartibles** por los que **compiten** los procesos. **Exclusión Mutua**.
- Los procesos concurrentes **cooperan** compartiendo información. **Condición de sincronización**.
- Las condiciones a cualquier solución satisfactoria fue establecida por Dijkstra en los años '60.
- En 1965 Dijkstra presenta el algoritmo del matemático holandés *Dekker*, como solución al problema de la exclusión mutua para dos procesos.

# 1. Introducción

- Posteriormente *Dijkstra* (1965) presenta una generalización del algoritmo de *Dekker* para n procesos.
- Diversas optimizaciones dan lugar en 1972 al algoritmo de Eisenberg-McGuire [Eisenberg an Mcguire, 1972].
- La dificultad de la solución queda patente con la publicación en los años '60 de alguna solución incorrecta, como fue el algoritmo de Hyman en 1966.

# 2. Tipos de sincronización y su solución

- **Exclusión mutua**: Acción de sincronización necesaria para que dos o más procesos puedan usar un recurso no compartible.
- **Sección crítica**: parte del código que utiliza un proceso en el acceso a un recurso no compartible, y por tanto, que debe ejecutarse en exclusión mutua.
- Cuando un proceso accede a una sección crítica, el resto de procesos que pretenden acceder a ella deben esperar a que ésta quede libre.
- Un proceso que abandona una sección crítica debe posibilitar el acceso a otro proceso que se encuentre esperando.

# 2. Tipos de sincronización y su solución

- Para garantizar la exclusión mutua se diseña un protocolo de entrada y otro de salida:

```
process P1
.......
Protocolo de entrada
Sección Crítica
Protocolo de salida
.......
```

```
process Pn
.......
Protocolo de entrada
Sección Crítica
Protocolo de salida
.......
```

# 2. Tipos de sincronización y su solución

- Para garantizar la exclusión mutua los protocolos de entrada y de salida deben satisfacer las siguientes condiciones:
  - **Exclusión mutua** Dos procesos no pueden estar a la vez en la sección crítica.
  - **Limitación de la espera.** Ningún proceso espera de forma indefinida.
  - **Progreso en la ejecución.** Un proceso que quiera acceder a la S.C lo hará si está libre.
- Además al ser código añadido para sincronización su ejecución debe ser rápida.

# 2. Tipos de sincronización y su solución

- **Condición de sincronización**: propiedad requerida de que un proceso no realice un evento hasta que otro proceso haya emprendido una acción determinada.
- Los mecanismos para resolver estos dos problemas son:
  - Inhibición de interrupciones
  - Espera ocupada (busy waiting)
  - Semáforos
  - Regiones Críticas Condicionales
  - Monitores

**Soluciones basadas en memoria compartida**

  - Operaciones de paso de mensajes (send/receive)
  - Invocaciones remotas

**Soluciones basadas en paso de mensajes**

# 2. Tipos de sincronización y su solución

- **Inhibición de interrupciones.** En un sistema con un procesador, la concurrencia se consigue mediante interrupciones (de reloj o de E/S).

```
process P1
.......
inhibir interrupciones
Sección Crítica
habilitar interrupciones
.......
```

```
process Pn
.......
inhibir interrupciones
Sección Crítica
habilitar interrupciones
.......
```

- Los procesos no tienen la posibilidad de inhibir las interrupciones.
- Sólo es factible para sistemas con un solo procesador.

# 3. Soluciones de Espera Ocupada

- Implementan la sincronización basándose en que un proceso **espera** comprobando de forma continua el valor de una variable, y por tanto manteniendo **ocupada** la CPU.
- Según las operaciones empleadas se distinguen entre:
  - **Soluciones software**. Las únicas instrucciones atómicas consideradas son las de leer/escribir direcciones de memoria.
  - **Soluciones hardware**: Se usan instrucciones especiales para llevar a cabo una serie de acciones.

# 3. Soluciones de Espera Ocupada

- Solución software al problema:
  - *Sea P1, P2, .. , Pn un conjunto de procesos que se suponen cíclicos; el código de cada uno de ellos contiene una sección crítica sobre el mismo recurso no compartible.*
  - *La exclusión mutua está asegurada mediante dos secuencias de instrucciones que enmarcan la sección crítica de cada proceso: una determinada* ***protocolo de entrada****, que comprueba una condición para autorizar la entrada a la sección critica, y otra denominada* ***protocolo de salida*** *que deberá indicar que se ha terminado de ejecutar dicha sección crítica.*

# 3. Soluciones de Espera Ocupada

- Solución al problema:
  - La notación usada para estos procesos es:

```
process Pi
repeat
    Protocolo de entrada
    Sección Crítica_i
    Protocolo de salida
    Resto_i
forever
```

  - Las únicas instrucciones atómicas que se consideran son las de leer y escribir en posiciones de memoria a bajo nivel (load-store)

# 3. Soluciones de Espera Ocupada

- Además:
  - Un proceso puede pararse por cualquier motivo en zona no crítica de instrucciones, pero **no le estará permitido parar durante la ejecución de los protocolos o de la sección crítica**.
  - El programa tampoco debe bloquearse. Si varios procesos del programa están intentando entrar en sus secciones críticas, **no se producirá un bloqueo**, sino que alguno de ellos conseguirá entrar en ella eventualmente. No obstante, si ninguno de los procesos tiene éxito en pasar del protocolo de entrada, diremos que el programa se ha bloqueado.

# 3. Soluciones de Espera Ocupada

- Además:
  - Tampoco debe haber permanencia indefinida de ninguno de los procesos en su protocolo de entrada, sino que en algún instante **cada proceso tendrá éxito en el acceso a su región crítica**.
  - En ausencia de contenciones (circunstancias que lo retengan), **un único proceso que desee entrar en su sección crítica tendrá éxito y entrará en ella**.

# 3. Soluciones de Espera Ocupada

- Algoritmos no eficientes: Primer intento
  - Inicialmente **v** vale **sclibre**.

```
process P0
repeat
   /*protocolo de entrada*/
   while v = scocupada do;
   v := socupada;
   /* ejecuta la S.C */
   Sección Crítica0
   /*protocolo de salida*/
   v := sclibre
   /*resto del proceso*/
   Resto0
forever
```

```
process P1
repeat
   /*protocolo de entrada*/
   while v = scocupada do;
   v := socupada;
   /* ejecuta la S.C */
   Sección Crítica1
   /*protocolo de salida*/
   v := sclibre
   /*resto del proceso*/
   Resto1
forever
```

# 3. Soluciones de Espera Ocupada

- Algoritmos no eficientes: Segundo intento
  - La variable **turno** se inicializa a **0** o **1** indistintamente.

```
process P0
repeat
    while turno = 1 do;
    Sección Crítica0
    turno := 1
    Resto0
forever
```

```
process P1
repeat
    while turno = 0 do;
    Sección Crítica1
    turno := 0
    Resto1
forever
```

  - No satisface la condición de progreso en la ejecución.
  - Si un proceso falla o termina antes que el otro se produce espera ilimitada.

# 3. Soluciones de Espera Ocupada

- Algoritmos no eficientes: Tercer intento
  - `c0` y `c1` se inicializan a **fueraSC**

```
process P0
repeat
    while c1 = enSC do;
    c0 := enSC;
    Sección Crítica0
    c0 := fueraSC;
    Resto0
forever
```

```
process P1
repeat
    while c0 = enSC do;
    c1 := enSC;
    Sección Crítica1
    c1 := fueraSC;
    Resto1
forever
```

  - No garantiza la exclusión mutua.

# 3. Soluciones de Espera Ocupada

- Algoritmos no eficientes: Cuarto intento
  - **c0** y **c1** se inicializan a **fueraSC**

```
process P0
repeat
    c0 := quiereentrar;
    while c1 = quiereentrar do;
    Sección Crítica0
    c0 := fueraSC;
    Resto0
forever
```

```
process P1
repeat
    c1 := quiereentrar;
    while c0 = quiereentrar do;
    Sección Crítica1
    c1 := fueraSC;
    Resto1
forever
```

  - Garantiza la exclusión mutua.
  - Produce un ***livelock***

# 3. Soluciones de Espera Ocupada

- Algoritmos no eficientes: Quinto intento
  - **c0** y **c1** se inicializan a **fueraSC**

```
process P0
repeat
   c0 := quiereentrar;
   while c1 = quiereentrar do
     begin
      c0 := fueraSC;
      /*realiza una espera*/
      c0 := quiereentrar;
     end
   Sección Crítica0
   c0 := fueraSC;
   Resto0
forever
```

```
process P1
repeat
   c1 := quiereentrar;
   while c0 = quiereentrar do
     begin
      c1 := fueraSC;
      /*realiza una espera*/
      c1 := quiereentrar;
     end
   Sección Crítica1
   c1 := fueraSC;
   Resto1
forever
```

No garantiza el acceso en tiempo finito. Imposible conocer la eficiencia

# 3. Soluciones de Espera Ocupada

- Algoritmo de Dekker (1965)
  - **c0** y **c1** se inicializan a **fueraSC.** El valor de **turno** no influye.

```
process P0
repeat
  c0 := quiereentrar;
  while c1 = quiereentrar do
   if turno = 1 them
    begin
     c0 := fueraSC;
     while turno = 1 do;
     c0 := quiereentrar;
    end
  Sección Crítica0
  turno := 1
  c0 := fueraSC;
  Resto0
forever
```

```
process P1
repeat
  c1 := quiereentrar;
  while c0 = quiereentrar do
   if turno = 0 them
    begin
     c1 := fueraSC;
     while turno = 0 do;
     c1 := quiereentrar;
    end
  Sección Crítica1
  turno := 0
  c1 := fueraSC;
  Resto1
forever
```

# 3. Soluciones de Espera Ocupada

- Algoritmo de Peterson (1981)
  - **c0** y **c1** se inicializan a **fueraSC.** El valor de **turno** no influye.

```
process P0
repeat
  c0 := quiereentrar;
  turno := 1;
  while (c1 = quiereentrar)
            and (turno = 1) do;
  Sección Crítica0
  c0 := fueraSC;
  Resto0
forever
```

```
process P1
repeat
  c1 := quiereentrar;
  turno := 0;
  while (c0 = quiereentrar)
            and (turno = 0) do;
  Sección Crítica1
  c1 := fueraSC;
  Resto1
forever
```

# 3. Soluciones de Espera Ocupada

- Algoritmo incorrecto de Hyman (1966)
  - **c0** y **c1** se inicializan a **fueraSC.** El valor de **turno** no influye.

```
process P0
repeat
 c0 := quiereentrar;
 while turno <> 0 do
  begin
   while c1 = quiereentrar do;
   turno := 0;
  end
  Sección Crítica0
  c0 := fueraSC;
  Resto0
forever
```

```
process P1
repeat
 c1 := quiereentrar;
 while turno <> 1 do
  begin
   while c0 = quiereentrar do;
   turno := 1;
  end
  Sección Crítica1
  c1 := fueraSC;
  Resto1
forever
```

# Algoritmo de Eisenberg-McGuire (1972)

```
process Pi
repeat
  repeat
    indicador[i]:= quiereentrar;
    j := indice; //indice es el id del proceso que tiene el turno
    while (j <> i) begin
      if indicador[j] <> fueraSC then j:= indice
      else j:= (j+1) mod n
    end
    indicador[i] := enSC;
    j := 0;
    while ((j<n) and ((j=i) or (indicador[j] <> enSC))) j:=j+1;
  until ((j>=n) and ((indice=i) or (indicador[indice]= fueraSC)));
  indice : = i;
  Sección Critica
  j := (indice+1) mod n;
  while (indicador[j]= fueraSC) j:=(j+1) mod n;
  indice = j;
  indicador[i] := fueraSC;
  Resto0
forever
```

# 3. Soluciones de Espera Ocupada

- Algoritmo de Lamport (1974)
  - Es posible usarlo en entornos distribuidos

```
process Pi
repeat //inicialmente numero[i]=0 -> el proceso no tiene numero
 c[i] := cognum;
 numero[i]:= 1+max(numero[0],...,numero[n-1]);
 c[i] := nocognum;
 for j=0 to n-1 do
  begin
   while (c[j] = cognum) do;
   while ((numero[j]<>0) and ((numero[i],i)>(numero[j],j))) do;
  end
 Sección Crítica1
 numero[i]=0;
 Resto_i
forever
```

# 4. La realidad del hardware

- Los algoritmos anteriores son correctos conceptualmente, pero **no funcionan con los procesadores actuales**.
- Se necesitan instrucciones especiales del procesador para hacer que los algoritmos funcionen.
- Esto se debe a que los fabricantes usan diversas técnicas para **mejorar el rendimiento de los procesadores**:
  - Múltiples niveles de caché
  - Buffers de escritura
  - Segmentación (*Pipelines*)
  - Uso de varios núcleos

# 4. La realidad del hardware

- Los procesadores modernos no garantizan que las instrucciones se ejecuten en el mismo orden que están escritas. No aseguran la *consistencia secuencial* de acceso a memoria.
- Las tres razones que pueden provocar la violación de la consistencia secuencial son:
  - Optimizaciones del compilador
  - Incoherencia de caché de RAM en multiprocesadores
  - Ejecución fuera de orden

# 4. La realidad del hardware

- **Optimizaciones del compilador**
  - Los compiladores pueden optimizar el código, cambiando el orden de ejecución o usar registros como almacenes de variables antes de copiarlos en la RAM.
  - Para evitar que una variable compartida no se vuelque a memoria, en C y Java se puede usar el modificador `volatile`

```
volatile int contador=0;
```

# 4. La realidad del hardware

- **Caché de RAM en multiprocesadores**
  - Los procesadores usan **jerarquía de caché de hasta 3 niveles** para reducir el tiempo de acceso a memoria.
  - Normalmente la L1 y L2 están integradas en el núcleo, y la L3 es compartida por varios núcleos en un mismo chip.
  - La cache almacena una linea de RAM (de 64 a 256 bits). Al acceder a una posición de RAM se almacena toda la linea en caché para futuros accesos.
  - Las **cachés modificadas son marcadas para volcarse posteriormente** (*write-back*), o directamente a la RAM (write-through)
  - Las arquitecturas multiprocesadores (SMP) deben **mantener la coherencia de las copias de caché**.

# 4. La realidad del hardware

- **Ejecución fuera de orden**
  - Los **procesadores re-ordenan las instrucciones** para ahorrar ciclos de CPU, por ejemplo, por tener cargados los registros o en los ***pipelines***, por tener la instrucción posterior decodificada.
  - Los procesadores no aseguran la consistencia secuencial del programa, pero usan **dependencias causales débiles** (*weak dependencies*) de acceso a memoria.
  - Por ejemplo, estas instrucciones se pueden ejecutar en cualquier orden que mantenga la dependencia entre la asignación de a y la de c:

```
a = x;
b = y;
c = a * 2
```

```
a = x;
c = a * 2
b = y;
```

```
b = y;
a = x;
c = a * 2
```

# 4. La realidad del hardware

- **Ejecución fuera de orden**
  - Las dependencias causales débiles funcionan bien en procesos aislados, pero en **procesos concurrentes no son capaces de detectar dichas dependencias.**
  - Por ejemplo, el algoritmo de Peterson, podría ejecutarse de la siguiente forma, y cumpliría las dependencias causales débiles:

```
process P0
repeat
   turno := 1;
   while (c1 = quiereentrar) and (turno = 1) do;
   c0 := quiereentrar;
   Sección Crítica0
   c0 := fueraSC;
forever
```

# 4. La realidad del hardware

- **Ejecución fuera de orden**
  - La ejecución fuera de orden es el principal problema de que fallen los algoritmos de exclusión mutua.
  - Para corregirlos, se debe pedir al procesador que respete el orden de acceso a memoria en los segmentos de programa críticos, y esto se hace mediante las **barreras de memoria (fences o barriers)**.
  - Una instrucción de barrera indica al procesador que:
    - Ejecute todas las lecturas y escrituras previas a la barrera
    - Ninguna operación de lectura/escritura posterior a la barrera se ejecute antes que ésta.

# 5. Soluciones Hardware

- Las soluciones previas sólo permitían realizar de forma atómica las operaciones básicas de acceso a memoria.
- Hay procesadores que proporcionan instrucciones para llevar a cabo varias acciones de forma indivisible.
- Estas instrucciones especiales pueden usarse para resolver el problema de la exclusión mutua con espera ocupada.
- Veremos estas instrucciones de forma genérica, y no para una máquina en particular.

# 5. Soluciones Hardware

- Instrucción **exchage**
- La instrucción **exchange(r,m)** intercambia el contenido de las posiciones de memoria **r** y **m** de forma atómica.
- La implementación del protocolo sería:
  - **m** inicialmente vale 1 y los $r_i$ valen 0

```
process P0
repeat
    repeat
        exchange(r0,m)
    until r0 = 1;
    Sección Crítica0
    exchange(r0,m)
    Resto0
forever
```

```
process P1
repeat
    repeat
        exchange(r1,m)
    until r1 = 1;
    Sección Crítica1
    exchange(r1,m)
    Resto1
forever
```

# 5. Soluciones Hardware

- Instrucción de **decremento**
- La instrucción **`subc(r,m)`** decrementa en 1 el contenido de **`m`** y copia el resultado en **`r`** de forma atómica.
- La implementación del protocolo sería:
  - **`m`** inicialmente vale 1

```
process P0
repeat
   repeat
      subc(r0,m)
   until r0 = 0;
   Sección Crítica0
   m := 1;
   Resto0
forever
```

```
process P1
repeat
   repeat
      subc(r1,m)
   until r1 = 0;
   Sección Crítica1
   m := 1;
   Resto1
forever
```

# 5. Soluciones Hardware

- Instrucción de **incremento**
- La instrucción **addc(r,m)** incrementa en 1 el contenido de **m** y copia el resultado en **r** de forma atómica.
- La implementación del protocolo sería:
  - **m** inicialmente vale -1

```
process P0
repeat
   repeat
      addc(r0,m)
   until r0 = 0;
   Sección Crítica0
   m := -1;
   Resto0
forever
```

```
process P1
repeat
   repeat
      addc(r1,m)
   until r1 = 0;
   Sección Crítica1
   m := -1;
   Resto1
forever
```

# 5. Soluciones Hardware

- Instrucción **`testset`**
- La instrucción **`testset(m)`** realiza la siguiente secuencia de acciones de forma atómica.
  - Comprueba el valor de la variable **`m`**.
  - Si el valor es 0 lo cambia a 1 y devuelve como resultado **`true`**.
  - En otro caso no modifica el valor y devuelve **`false`**.
- La implementación del protocolo sería: (**`m`** inicialmente vale 0)

```
process P0
repeat
   repeat until testset(m);
   Sección Crítica0
   m := 0;
   Resto0
forever
```

```
process P1
repeat
   repeat until testset(m);
   Sección Crítica1
   m := 0;
   Resto1
forever
```

# 5. Soluciones Hardware

- Instrucción **testset**

```
process Pi
repeat
  esperando[i] := true;
  llave := false;
  repeat
    llave := testset(m)
  until (not esperando[i]) or (llave);
  esperando[i]=false;
  Sección Crítica_i
  j := (i+1) mod n;
  while (j<>i) and (not esperando[j]) do j:=(j+1) mod n;
  if (j=i) then m:=0
  else esperando[j] := false;
  Resto_i
forever
```

# 5. Soluciones Hardware

- **Instrucciones atómicas en JAVA**
  - Java proporciona, a través de su paquete **`java.util.concurrent.atomic`** un conjunto de objetos que representan a distintos tipos de datos y sobre los cuales se han definido operaciones que serán realizadas de forma atómica, es decir, sin posibilidad de que sean interrumpidas entre su comienzo y su fin.
  - Algunos de sus componentes son:
    - **`AtomicBoolean.`** Un booleano que puede ser actualizado atómicamente.
    - **`AtomicInteger.`** Un entero que puede ser actualizado atómicamente.
    - **`AtomicIntegerArray.`** Un array de enteros cuyos elementos pueden ser actualizados atómicamente.
    - **`AtomicLong`** Un **`long`** que puede ser actualizado atómicamente.
    - **`AtomicLongArray`** Un array de **`long`** cuyos elementos pueden ser actualizados atómicamente.

# 5. Soluciones Hardware

- **Instrucciones atómicas en JAVA**
  - A modo de ejemplo, algunos de los métodos que es posible usar sobre un `AtomicInteger` son:
    - `int get()`
    - `void set(int newValue)`
    - `int addAndGet(int delta)`
    - `int decrementAndGet()`
    - `int incrementAndGet()`
    - `boolean compareAndSet(int expect, int update).` Establece el valor a `update` si el valor actual se corresponde con `expect`, devolviendo `true`. En caso contrario no hace nada y devuelve `false`.

# 5. Soluciones Hardware

- **Instrucciones atómicas en JAVA**
  - La equivalencia con las intrucciones vistas anteriormente es inmediata:
    - `AtomicInteger m = new AtomicInteger(0);`
      - `subc(r,m)` ↔ `r = m.decrementAndGet()`
      - `addc(r,m)` ↔ `r = m.incrementAndGet()`
      - `testset(m)` ↔ `m.compareAndSet(0,1)`

# Ejercicios

**Comprobar la validez del algoritmo**

**Inicialmente C1=1 y C2=1**

```
process P1;
begin
 repeat
  repeat
   C1:=1-C2
  until C2 <> 0;
  {SECCIÓN CRITICA}
  C1:=1;
 {Resto1}
forever
end;
```

```
process P2;
begin
 repeat
  repeat
   C2:=1-C1
  until C1 <> 0;
  {SECCIÓN CRITICA}
  C2:=1;
  {Resto1}
 forever
end;
```

# Ejercicios

**Solución**

**Inicialmente C1=1 y C2=1**

```
(1)  Repeat C1:=1-C2 until C2 <> 0;
       {SECCIÓN CRITICA}
(3)  C1:=1;
```

```
(2) Repeat C2:=1-C1
(4)   until C1 <> 0;
       {SECCIÓN CRITICA}
      C2:=1;
```

**En (1) C1=0 y C2=1, entonces P1 entra en SC**
**En (2) C2=1**
**En (3) C1=1**
**En (4) P2 entra en SC sin hacer c2=0, con lo cual P1 entra también en SC**

# Ejercicios

**Comprobar la validez del algoritmo**

**Inicialmente C1=1 y C2=1**

```
process P1;
begin
 repeat
  C1=0;
  while C2=0 do begin
   C1:=1;
   while C2=0 do;
   C1:=0;
  end
  {SECCIÓN CRITICA}
  C1:=1;
 {Resto1}
forever
end;
```

```
process P2;
begin
 repeat
  C2=0;
  while C1=0 do begin
   C2:=1;
   while C1=0 do;
   C2:=0;
  end
  {SECCIÓN CRITICA}
  C2:=1;
 {Resto1}
forever
end;
```

# Ejercicios

**Comprobar la validez del algoritmo**

```
Inicialmente C1=1 y C2=1
```

**Solución: Ejecutando alternativamente una instrucción de cada proceso, ambos se quedan indefinidamente ejecutando su protocolo de entrada sin poder acceder a la SC.**