# Concurrencia y Distribución

Dr. Arno Formella
**Universidad de Vigo**
**Departamento de Informática**
**Área de Lenguajes y Sistemas Informáticos**
**E-32004 Ourense**

`http://www.ei.uvigo.es/~formella`
Email: formella@ei.uvigo.es

5 de junio de 2001

# Índice General

# 1 Curso

| | |
|---|---|
| **Teoría:** | los lunes, 16-18 horas, Aula 3.2 |
| **Prácticas:** | dos grupos, lunes, 18-20 horas y 20-22 horas, Lab. SO-B |
| **Asignaturas vecinas:** | todo sobre programación, sistemas operativos, procesamiento paralelo, sistemas de tiempo real, diseño de software |
| **Prerequisitos:** | Java, C, programación secuencial |
| **Examen:** | escrito, final del curso, teoría y práctica juntos |
| **Créditos:** | 6 |
| **Literatura:** | mira Bibliografía (Section 22) |

# 2 Objetivos

- conocer los principios y las *metodologías* de la programación concurrente y distribuida
- conocer las principales *dificultades* en realizar programas concurrentes y distribuidos
- conocer *herramientas* existentes para afrontar la tarea de la programación concurrente y distribuida
- conocer el concepto de concurrencia en *Java*

# 3 Sobre este documento

Este documento crecerá durante el curso, *ojo, no necesariamente solamente al final.*

Los ejemplos de programas y algoritmos serán en inglés.

Uso de código de colores en este documento:

- algoritmo
- código fuente

# 4 Introducción

No existe una clara definición de programación concurrente en la literatura. No se puede separar fácilmente el término programación concurrente del término programación en paralelo.

## 4.1 Aclaración

Una tendencia según mi opinión es:

- la programación concurrente se dedica más a *desarrollar* y *aplicar* conceptos para el uso de recursos en paralelo
- la programción en paralelo se dedica más a *solucionar* y *analizar* problemas bajo el conceptos del uso de recursos en paralelo

Otra posibilidad de separar los términos es:

- un programa concurrente define las acciones que se pueden ejecutar simultaneamente
- un programa paralelo es un programa concurrente diseñado de estar ejecutado en hardware paralelo
- un programa distribuido es un programa paralelo diseñado de estar ejecutado en hardware distribuido, es decir, donde varios procesadores no tengan memoria compartida

Intuitivamente, todos tenemos una idea de lo que significa el concepto de concurrencia.

**Ejemplo:**

- sumamos algunos números
- si lo hace uno solo ...
- si lo hacemos juntos ...

## 4.2 Procesos

Es decir, subdividimos la tarea en trozos que se pueden resolver en paralelo. Dichos trozos llamamos *procesos*. Es decir, un proceso (un nuestro contexto) es

- una secuencia de instrucciones o sentencias que
- se ejecuta secuencialmente en un procesador

En la literatura, sobre todo en el ámbito de sistemas operativos, existen también los conceptos de hilos ("threads") y de tareas ("tasks" o "jobs") que son parecidos al concepto de proceso, aún se distinguen en varios aspectos. En nuestro contexto no distinguimos mucho más.

Solo destacamos el concepto de hilo que se usa casi siempre en la programación moderna. Un programa multi-hilo intercala varias secuencias de instrucciones que usan los mismos recursos bajo el techo de un sólo proceso en el sentido de unidad de control del sistema operativo, que no se debe confundir con nuestro concepto abstracto de proceso.

Un programa secuencial consiste en un solo proceso.

En un programa concurrente trabaja un conjunto de procesos en paralelo cuales cooperan para resolver un problema.

## 4.3 Aplicación

¿Cuándo se usan programas concurrentes?

- cuando nos dé la gana, lo principal es: *solucionar el problema*, y
- cuando los recursos lo permitan, lo principal es: *conocer las posibilidades y herramientas*

### 4.3.1 Indicadores

¿Cuáles son indicadores que sugieren un programa concurrente?

- el problema consiste de forma natural en gestionar eventos
- el problema consiste en proporcionar un alto nivel de disponibilidad, es decir, nuevos eventos recién llegados requieren una respuesta rápida (disponibilidad, "availability")
- el problema exige un alto nivel de control, es decir, se quiere terminar o suspender tareas una vez empezadas (controlabilidad, "controllability")
- el problema tiene que cumplir restricciones temporales
- el problema requiere que varias tareas se ejecutan (quasi) simultaneamente (programación reactiva, "reactive programming")
- se quiere ejecutar un programa más rápido y los recursos están disponibles (explotación del paralelismo, "Explotation of parallelism")
- el problema consiste en simular objetos reales con sus comportamientos y interacciones indeterminísticos (objetos activos, "active objects")

Eso implica que hay que tomar decisiones qué tipo y qué número de procesos se usa y en qué manera deben interactuar.

### 4.3.2 Recursos

Entro otros, posibles recursos son

- procesadores
- memoria
- dispositivos periféricos (p.e., impresoras, líneas telefónicas) sobre todo de entrada y de salida
- estructuras de datos con sus contenidos

### 4.3.3 Ejemplos

Existen ejemplos de problemas que por su naturaleza deben diseñarse como programas concurrentes:

- sistemas operativos
  - soportar operaciones quasi-paralelas
  - proveer servicios a varios usuarios a la vez (sistemas multi-usuario, sin largos tiempos de espera)
  - gestionar recursos compartidos (p.e., sistemas de ficheros)
  - reaccionar a eventos no predeterminados
- sistemas de tiempo real
  - necesidad de cumplir restricciones temporales
  - reaccionar a eventos no predeterminados
- sistemas de simulación
  - el sistema por simular ya dispone de modulos que funcionan en forma concurrente
  - el flujo del control no sigue un patrón secuencial
- booking systems
  - las aplicaciones se ejecutan en diferentes lugares
- sistemas de transacciones
  - se tiene que esperar la terminación de una transacción antes de poner en marcha la siguiente
  - varias transacciones en espera pueden compartir por ser ejecutado con diferentes prioridades
- controladores de tráfico aéreo
  - el sistema tiene que estimar el futuro próximo sin perder la capacidad de reaccionar rápidamente a cambios bruscos
- sistemas de comunicación
  - la interfaz al usuario requiere un alto nivel de disponibilidad y controlabilidad
  - en la época de la comunicación digital, todos queremos usar la red (o bién alámbrica o bién inalámbrica) al mismo tiempo sin notar que habrá más gente con las mismas ambiciones
- sistemas tolerantes a fallos
  - se vigila de forma concurrente el funcionamiento correcto de otra aplicación

En particular, resultará escencial el desarrollo de un programa concurrente cuando la concurrencia de activades es un aspecto inerto del problema a resolver.

Programadores modernos tienen que saber como escribir programas que manejan multiples procesos.

## 4.4 Implementación

Enfocamos solamente en programas escritos en lenguajes imperativos con concurrencia, comunicación y sincronización explícita.

Como cualquier otra tarea de programación estamos confrontados con los problemas de

- la especificación del programa,
- el diseño del programa,
- la codificación del programa, y
- la verificación del programa.

## 4.5 Crítica

Entre las ventajas de la programación concurrente se suele encontrar

- mejor rendimiento, es decir, se espera
  - que el programa se ejecute más rápido
  - que el programa use los recursos de mejor manera, p.e., no deja recursos disponibles sin usar durante mucho tiempo
  - que el programa se desarrolle más fácil
  - que el programa refleje el modelo del problema real

Sin embargo, también existen desventajas:

- se pierde tiempo en sincronizar procesos y comunicar datos entre ellos
- en el caso de multiplexeo de procesos/hilos se pierde tiempo en salvar información sobre el contexto
- los procesos pueden esperar a acciones de otros procesos, eso puede resultar en un bloqueo ("deadlock") de algún proceso, en el peor caso se daría como resultado que se produce ningún progreso en el programa
- hay que buscar estrategias eficientes para distribuir el trabajo entre los diferentes procesadores ("efficient load balancing")
- hay que buscar estrategias eficientes para distribuir los datos entre los diferentes procesadores ("efficient data distribution")
- en muchas situaciones hay que buscar un compromiso entre tiempo de ejecución y uso de recursos
- el desarrollo de programas concurrentes es mucho más complejo que el desarrollo de programas secuenciales
- la depuración de programas concurrentes es *muy difícil*

# 5 Repaso: programación secuencial

Asumimos que tengamos solamente las operaciones aritméticas sumar y subtraer disponibles y queremos multiplicar dos números positivos.

Un posible algoritmo sequencial que multiplica el número `p` con el número `q` produciendo el resultado `r` es:

```
Initially:  p is set to positive number
            q is set to positive number

a: set r to 0
b: loop
c:   if q equal 0 exit
d:   set r to r+p
e:   set q to q-1
f: endloop
g: ...
```

¿Cómo se comprueba si el algoritmo es correcto?

Primero tenemos que decir que significa correcto.

El algoritmo (secuencial) es correcto si

- una vez haber llegado a la instrucción `g:` el valor de la variable `r` contiene el producto de los valores de las variables `p` y `q` (se refiere a sus valores llegando a la instrucción `a:`)
- se llega a la instrucción `g:` en algún momento

Tenemos que saber que las instrucciones atómicas son correctos, es decir, sabemos exactamente su significado.

Luego usamos el concepto de inducción para comprobar el bucle.

# 6 Primer algoritmo concurrente

```
Initially:  p is set to positive number
            q is set to positive number

a: set r to 0
 P0                         P1
b: loop                       loop
c:   if q equal 0 exit          if q equal 0 exit
d:   set r to r+p               set r to r+p
e:   set q to q-1               set q to q-1
f: endloop                    endloop
g: ...
```

El algoritmo es indetermístico en el sentido que no se sabe de antemano en qué orden se van a ejecutar las instrucciones, o más preciso, cómo se van a intercalar las instrucciones de ambos procesos.

En este momento anotamos que para un algoritmo concurrente las siguientes propriedades son escenciales:

**seguridad:** nunca pasa nada malo, es decir, una cierta condición se cumple siempre ("safety property")

**vivacidad:** a veces pasa algo bueno, es decir, al fin y al cabo pasa algo o en algún momento en el futuro se cumple una cierta condición ("liveness property")

El indeterminismo puede provocar situaciones que resultan en errores transitorios, es decir, el fallo ocurre solamente si las instrucciones se ejecutan en un orden específico.

**Ejemplo:** multiplicación

Un programa concurrente es correcto si el resultado observado (y esperado) no depende del orden (dentro de todos los posibles órdenes) en el cual se ejecute las instrucciones.

Para comprobar si un programa concurrente es incorrecto basta con encontrar una intercalación de instrucciones que resulta en un fallo.

Para comprobar si un programa concurrente es correcto hay que comprobar que no se produce ningún fallo en ninguna de las intercalaciones posibles.

El número de posibles intercalaciones de los procesos en un programa concurrente crece exponencialmente con el número de unidades que maneja el planificador. Por eso es prácticamente imposible comprobar con mera enumeración si un programa concurrente es correcto baja todas las ejecuciones posibles.

En la argumentación hasta ahora fue muy importante que las instrucciones se han ejecutado de forma atómica, es decir, sin interrupción ninguna.

Por ejemplo, se observa una gran diferencia si el procesador trabaja directamente en memoria o si trabaja con registros:

```
P1:  inc N
P2:  inc N

P2:  inc N
P1:  inc N
```

Se observa: las dos intercalaciones posibles producen el resultado correcto.

```
P1:  load  R1,N
P2:  load  R2,N
P1:  inc   R1
P2:  inc   R2
P1:  store R1,N
P2:  store R2,N
```

Es decir, existe una intercalación que produce un resultado falso.

Eso implica directamente: no se puede convertir un programa multi-hilo en un programa distribuido sin analizar el concepto de memoria común detenidamente.

# 7 Arquitecturas que soportan la concurrencia

Se suele distinguir concurrencia

- de grano fino
  es decir, se aprovecha de la ejecución de operaciones concurrentes al nivel del procesador (hardware)
- de grano grueso
  es decir, se aprovecha de la ejecución de procesos o aplicaciones al nivel del sistema operativo o al nivel de la red de ordenadores

Entre ambos extremos caben muchos granos intermedios.

Una clasificación clásica de ordenadores paralelos es:

- SIMD (single instruction multiple data)
- MISD (multiple instruction single data)
- MIMD (multiple instruction multiple data)

No hace falta que los procesos necesariamente se ejecutan en hardware distinto (multi-programación o "multi-programming"). Se puede aprovechar de la posibilidad de multiplexar varios procesos en un solo procesador (multi-procesamiento o "multi-processing"). El sistema operativo (muchas veces con la ayuda de hardware específico) realiza la ejecución de varios procesos de forma quasi-paralelo distribuyendo el tiempo disponible a las secuencias diferentes ("time-sharing system").

En este caso es imprecindible para el análisis del programa que el mecanismo de conmutación sea independiente del programa concurrente, aún puede tomar sus decisiónes analizando las aplicaciones. En el caso contrario hay que incluir dicho mecanismo en el análisis del programa concurrente. Al desarrollar un programa concurrente, no se debe asumir ningún comportamiento específico del planificador (siendo la unidad que realiza la conmutación de los procesos).

**Ejemplo:** Ya vimos que si se asuma una intercalación perfecta, el algoritmo de multiplicación resulta correcta.

Existen muchos tipos de arquitecturas de ordenadores. Algunos son diseñados especialmente para la ejecución de programas concurrentes o paralelos.

Sin embargo, no hace falta que se ejecuta el programa en unidades similares para obtener concurrencia. Concurrencia está presente también en sistemas heterógenos, p.e., cuales solapan el trabajo de entrada y salida con el resto de las tareas.

La comunicación y sincronización (Section 8) entre procesos funciona mediante una memoria común ("shared memory") cual pueden acceder todos los procesadores a la vez o mediante el intercambio de mensajes usando una red conectando los diferentes procesadores o ordenadores ("distributed processing").

También existen mezclas de todo tipo de estos conceptos, p.e., un sistema que use multi-procesamiento con hilos y procesos en cada procesador de un sistema distribuido simulando una memoria común al nivel de la aplicación.

# 8 Comunicación y sincronización

Programas concurrentes necesitan algún tipo de comunicación entro los procesos. Hay dos razones principales:

1. los procesos compiten para obtener acceso a recursos compartidos,
2. los procesos quieren intercambiar datos.

Herramientas de programación suelen proporcionar solamente una de las dos posibilidades.

En ambos casos hace falta un método de sincronización entre los procesos que quieren comunicarse entre ellos. Al nivel del programador existen tres variantes como realizar las interacciones entre procesos:

1. usar memoria compartida,
2. mandar mensajes,
3. lanzar procedimientos remotos.

Con la sincronización condicional se protege un objeto (p.e, un trozo de memoria) para que ningún otro proceso puede acceder el objeto hasta que este se encuentre en un estado permitido.

Los objetivos requeridos para la sincronización de procesos son normalmente:

- exclusión mutua
- no bloqueo
- no espera innecesaria
- justo

La comunicación no tiene que ser síncrono en todos los casos. Existe también la forma asíncrona donde un proceso deja su mensaje en una estructura de datos compartida por los procesos. El proceso que ha mandado los datos puede seguir con otras tareas. El proceso que debe leer los datos lo hace en su momento.

Una comunicación entre procesos no tiene que ser fiable en todos los sistemas, existen varias posibilidades de fallos:

- se pierden mensajes
- se cambia el orden de los mensajes
- se añaden mensajes que nunca fueron mandados
- se modifican mensajes

# 9 Abstracción

Un programa concurrente se puede ver como un conjunto de

- procesos que consisten en secuencias de instrucciones atómicas (Section 9.1)
- cuyo tiempo de ejecución es indivisible y finito y
- hasta que no se quiera hacer una estimación del tiempo real no se asume nada sobre dicho tiempo finito y finalmente.

No se puede asumir de antemano ninguna información sobre el tiempo de ejecución de un proceso (especialmente relativo respeto a otros procesos). Desde el punto de vista abstracto no importa si los procesos se ejecutan en unidades independientes o si algún planificador distribuye los procesos multiplexando un solo procesador. De todas maneras, las características del planificador del sistema no están conocidas al programador.

## 9.1 Instrucciones atómicas

Se considera instrucciones atómicas aquellas que están garantizados en cumplir correctamente independiente de otras instrucciones posiblemente ejecutado simultaneamente en el programa. A veces también se refiere a instrucciones atómicas cuando el procesador es capaz de re-instalar el estado justamente antes de haber empezado la ejecución de la instrucción cuando se haya producido una interrupción, es decir, el efecto de la instrucción es nulo.

Apuntamos la secuencia de instrucciones atómicas del programa como $(a_0, a_1, ..., a_n)$.

Un algoritmo secuencial impone un orden total en el conjunto de las instrucciones que establece mientras un algoritmo concurrente solamente especifica un orden parcial. Eso se puede visualizar con grafos: los nodos representan las instrucciones atómicas, las aristas indican si una instrucción debe seguir la otra.

Aquí debe venir figura seqorden.ps

Aquí debe venir figura conorden.ps

Eso implica que la ejecución del programa es indeterminística. Ejecutar el mismo programa varias veces, aún con los mismos datos de entrada, puede resultar en secuencias de instrucciones diferentes, incluso puede ocurrir que es imposible detectar en que orden se ejecutan las instrucciones en un caso real.

## 9.2 Funcionamiento correcto

Generalmente se dice que un programa es correcto si dado una entrada el programa produce los resultados deseados.

Más formal:

Sea $P(x)$ una propriedad de una variable $x$ de entrada (aquí el símbolo $x$ refleja cualquier conjunto de variables de entradas). Sea $Q(x, y)$ una propriedad de una variable $x$ de entrada y de una variable $y$ de salida.

Se define dos tipos de funcionamiento correcto de un programa:

**funcionamiento correcto parcial:** dado una entrada $a$, si $P(a)$ es verdadero, y si se lanza el programa con la entrada $a$, entonces si el programa termina habrá calculado $b$ y $Q(a, b)$ también es verdadero.

**funcionamiento correcto total:** dado una entrada $a$, si $P(a)$ es verdadero, y si se lanza el programa con la entrada $a$, entonces el programa termina y habrá calculado $b$ con $Q(a, b)$ siendo también verdadero.

Para un programa secuencial existe solamente un orden total de las instrucciones atómicas, mientras que para un programa concurrente puedan existir varios órdenes.

Aquí debe venir figura condifor.ps

Por eso se tiene que exigir:

**funcionamiento correcto concurrente:** un programa concurrente funciona correctamente si el resultado $Q(x, y)$ no depende del orden de las instrucciones atómicas entre todos los órdenes posibles.

### 9.3 Regiones críticas

Una región crítica es una secuencia de instrucciones que no debe ser interrumpida por otros procesos, es decir, se debe tratar una región crítica como una sola instrucción atómica.

No es suficiente que los recursos usados en una región crítica no se deben ser alterados por otros procesos, porque es posible que su valor o contenido en el momento de lectura no están validos; puede ser que estén en un estado transiente. Sin embargo, si los accesos concurrentes lean solamente pueden estar permitidos (más sobre el tema veremo más adelante).

Apuntamos regiones críticas con $r_0$, ..., $r_m$.

Normalmente se protege en los lenguajes de programación solamente el código directamente, los datos están protegidos indirectamente por su código de acceso.

Por ejemplo, en Java no se puede declarar una clase como `synchronized`, sino solamente sus métodos. Eso requiere mucha disciplina desde el lado del programador en cuanto a acceder variables críticas solamente con métodos adecuados.

## 10 Representaciónes usadas en la programación concurrente

Dentro de código fuente, la inicialización de procesos (o como se llama la unidad en cada caso) suele ser diferente dependiendo del lenguaje y entorno usado.

Existen, p.e., palabras reservadas como `coroutine` (rutina/procedimiento concurrente), `fork` (bifurcación), `cobegin` y `coend` (sentencia concurrente) para iniciar el trabajo en paralelo. Otras se usa para sincronizar, p.e., `resume`, o `join`.

En el caso de hilos, todos los lenguajes modernos suelen tener un amplio conjunto de objetos y/o métodos respectivamente funciones para su uso.

## 10.1 Diagramas de estados

Usamos el concepto de programación orientado a objectos. Al llegar un mensaje por procesar, el objeto puede cambiar de un estado a otro.

Aquí debe venir figura estado.ps

## 10.2 Representación gráfica

Existen dos formas de representar programas concurrentes gráficamente:

- diagrama de interacciones (de UML)

  Aquí debe venir figura diauml.ps

- grafo de dependencias entre procesos

  Aquí debe venir figura diagrafo.ps

- autómata de estados

  Aquí debe venir figura diaauto.ps

- diagrama de flujo de datos

  Aquí debe venir figura diaflujo.ps

## 10.3 Lógica temporal

Para comprobar si un algoritmo concurrente es correcto hace falta un formalismo que es capaz de incluir los efectos que un proceso puede causar en el estado de otro proceso a lo largo del tiempo.

La lógica temporal es una extensión de la lógica clásica que incluye en sus formulas el tiempo y proporciona una técnica formal de comprobar propriedades de programas concurrentes.

Entro los operadores se suele encontrar: *ahora (now), después (next), hasta (until), antes (previous), desde (since)* y adicionalmente *finalmente (eventually)* y *a-partir-de-ahora-en-adelante (henceforth)*.

Con estos operadores se puede producir, p.e., sentencias como

- si $p$ *ahora*, entonces *finalmente* $q$
- *después* de todos los $p_i$, va a ocurrir $q$

donde $p$ y $q$ son variables que describen el estado del programa.

### 10.4 Redes de Petri

# 11 Exclusión mutua a nivel bajo

Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control.

Un método es usar un tipo de cerrojo ("lock"). Si un proceso quiere usar el recurso, mira si el cerrojo está abierto, si lo está, cierra el cerrojo y usa el recurso. Una vez haber terminado, abre el cerrojo de nuevo. Si está cerrado el cerrojo, tiene que esperar o dedicarse a otra cosa que queda por hacer.

Notamos el recurso con una variable $r$ (que puede ser, p.e., una impresora o un fichero o una estructura de datos (cola o lista)).

Mientras un proceso está accedando el recurso $r$ ningún otro proceso debe interferir (especificamos más adelante este 'interferir' más preciso). El problema más grande ocurre cuando dos procesos intentan modificar el estado del recurso al mismo tiempo.

En otras palabras, el proceso está ejecutando una región crítica que usa el recurso $r$ y todos los demás procesos deben estár excluidos.

Desarrollo del algoritmo de Dekker y Peterson

## 11.1 Algoritmo de Dekker

Intentamos controlar un solo recurso $r$ que quieren usar dos procesos $P0$ y $P1$.

### 11.1.1 Primer intento

Usamos una variable `v` que nos indicará cual de los dos procesos tiene su turno.

```
  P0                                P1
a:  loop                              loop
b:    non-critical section              non-critical section
c:    wait until v equals P0            wait until v equals P1
d:    critical section using r          critical section using r
e:    set v to P1                       set v to P0
f:  endloop                           endloop
```

- Obviamente, los procesos pueden acceder el recurso solamente alternamente.
- Si un proceso se termina, el otro puede resultar bloqueado.

### 11.1.2 Segundo intento

Usamos para cada proceso una variable `v0` respectivamente `v1` que indica si el correspondiente proceso está usando el recurso.

```
P0                                P1
  non-critical section              non-critical section
  wait until v1 equals false        wait until v0 equals false
  set v0 to true                    set v1 to true
  critical section using r          critical section using r
  set v0 to false                   set v1 to false
```

- Ya no existe la situación de posible bloqueo.
- Sin embargo: el algoritmo no está seguro, porque los dos procesos pueden alcanzar sus secciones críticas simultáneamente.

El problema está escondido en el uso de las variables de control. `v0` se debe cambiar a verdadero solamente si `v1` sigue siendo falso.

### 11.1.3 Tercer intento

Cambiamos el lugar donde se modifica la variable de control:

```
P0                                P1
  non-critical section              non-critical section
  set v0 to true                    set v1 to true
  wait until v1 equals false        wait until v0 equals false
  critical section using r          critical section using r
  set v0 to false                   set v1 to false
```

- Está garantizado que no entren ambos procesos al mismo tiempo en sus secciones críticas.
- Pero se bloquean mutuamente en caso que lo intentan simultaneamente.

### 11.1.4 Cuarto intento

```
P0                                P1
  non-critical section              non-critical section
  set v0 to true                    set v1 to true
  repeat                            repeat
    set v0 to false                   set v1 to false
    set v0 to true                    set v1 to true
  until v1 equals false             until v0 equals false
  critical section using r          critical section using r
  set v0 to false                   set v1 to false
```

starvation: un proceso o unos proceses siguen con su trabajo pero otros nunca llegan a utilizar los recursos

livelock: es una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil

### 11.1.5 Quinto intento

1968

Al final obtenemos con el quinto intento el algoritmo de Dekker:

```
Initially:  v0 is equal false
            v1 is equal false
            v  is equal P0 o P1

P0                                P1
  non-critical section              non-critical section
  set v0 to true                    set v1 to true
  repeat                            repeat
    if v equals P1                    if v equals P0
      set v0 to false                   set v1 to false
      wait until v equals P0            wait until v equals P1
      set v0 to true                    set v1 to true
    fi                                fi
  until v1 equals false             until v0 equals false
  critical section using r          critical section using r
  set v0 to false                   set v1 to false
  set v to P1                       set v to P0
```

El algoritmo de Dekker resuelve el problema de exclusión mutua en el caso de dos procesos (con memoria común).

Existen algoritmos que resuelven el problema para $n$ procesos, p.e., el algoritmo de Lambert (algoritmo de la panadería), o el algoritmo de Peterson.

## 11.2 Algoritmo de Peterson

## 11.3 Algoritmo de Lambert

## 11.4 Ayuda con hardware

Si existen instrucciones más potentes en el microprocesador se puede realizar el exclusión mutua más fácil.

### 11.4.1 Test-and-set

La instrucción test-and-set (TAS) implementa una comprobación del contenido de una variable en la memoria al mismo tiempo que varía su contenido en caso que la comprobación daba exito.

Así se puede realizar la exclusión mutua con

```
Loop:
  TAS lockbyte           ; test-and-set operation
  BNZ LOOP               ; loop until we get access
  ... critical section
  CLR lockbyte           ; clear lock
```

En caso de un sistema multi-procesador hay que tener cuidado que la operación test-and-set esté realizada cerca de la memoria compartida.

### 11.4.2 Atomic-swap

# 12 Propriedades de programas concurrentes

## 12.1 Seguridad y vivacidad

Un programa concurrente puede fallar por varias razones cuales se puede clasificar entre dos grupos de propriedades:

**seguridad:** Esa propriedad inidca que no está pasando nada malo en el programa, es decir, el programa no ejecuta instrucciones que no debe hacer.

**vivacidad:** Esa propriedad inidca que está pasando continuamente algo bueno durante la ejecución. es decir, el programa consigue algún progreso en sus tareas.

Las propriedades de seguridad suelen ser unaas de las invariantes des programa que se tiene que introducir en las comprobaciones del funcionamiento correcto (p.e., mediante inducción).

Ejemplos de propiedades de seguridad:

- el algoritmo usado es correcto
- la exclusión mutua de regiones críticas

Ejemplos de propiedades de vivacidad:

- ningún proceso se muere por inanición
- si un proceso pide un recurso, lo consigue en algún momento
- los procesos no se bloquean mutuamente
- no se termina un proceso desde fuera sin razón (p.e. en Java, cada `stop()` es razonable)
- un proceso no queda dormido (p.e. en Java, cada `suspend()` tiene su `resume()` correspondiente)
- la conexión entre procesos es fiable

## 12.2 Justicia entre procesos

Si procesos compiten por acceso a recursos compartidos se puede definir varios conceptos de justicia:

**justicia débil:** si un proceso pide acceso continuamente, le está dado en algún momento del futuro

**justicia estricta:** si un proceso pide acceso infinitamente veces, le está dado en algún momento del futuro

**espera limitada:** si un proceso pide acceso una vez, le está dado antes de que otro proceso lo obtenga más de una vez

**espera ordenada en tiempo:** si un proceso pide acceso, le está dado antes de todos los procesos que lo han pedido más tarde

Los dos primeros conceptos no son muy prácticos porque dependen de términos infinitamente o el algún momento, sin embargo, pueden ser útiles en comprobaciones formales.

En un sistema distribuido la ordenación en tiempo no es tan fácil de realizar dado que la noción de tiempo no está tan claro.

Normalmente se quiere que todos los procesos manifestan algún progreso en su trabajo. Sin embargo, eso no es necesario en programas concurrentes; se puede vivir bién con algunos procesos "muertos", mientras no dan otros problemas para el controlador (p.e., como llenar las tablas limitadas del sistema operativo). Siempre existe la posibilidad que el trabajo asignado a un proceso está hecho por otro proceso dejando el primero en espera infinita.

## 12.3 Espera activa

El algoritmo de Dekker y sus primos provocan una espera activa de los procesos cuando quieren acceder un recurso compartido. Mientras están esperando a entrar en su región crítica no hacen nada más que comprobar el estado de alguna variable.

Normalmente no es aceptable que los procesos pertenezcan en estos bucles de espera activa porque se está gastando potencia del procesador inútilmente.

Un método mejor es suspender el trabajo del proceso y reanudar el trabajo cuando la condición necesaria se haya cumplido. Naturalmente dichos técnicas de control son más complejas en su implementación que la simple espera activa.

## 12.4 Espera infinita o inanición

En programas concurrentes es posible que un proceso nunca llega a hacer nada si el planificador o el control de los recursos compartidas respectivamente no permiten que el proceso pueda cumplir con sus pedidos. Es decir, el proceso está sumetida a una espera infinita, o en otras palabras, sufre una inanición.

Existen varias técnicas para evitar posible inanición:

- El acceso a recursos compartidos siempre sigue el orden FIFO, es decir, los procesos tienen acceso en el mismo orden en que han pedido vez.
- Se asigna prioridades a los procesos de tal manera que cuando más tiempo un proceso tiene que esperar más alto se pone su prioridad con el fin que en algún momento su prioridad es la más alta.

# 13 Exclusión mutua a nivel alto

El concepto de usar estructuras de datos al nivel alto libera al programador de los detalles de su implementación. El programador puede asumir que las operaciones están implementados correctamente y puede basar el desarrollo del programa concurrente en un funcionamiento correcto de las operaciones de los tipos de datos abstractos.

Las implementaciones concretas de los tipos de datos abstractos tienen que recurrir a las posibilidades descritas arriba.

## 13.1 Semáforos

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso de manera exclusiva cuando varios procesos están competiendo.

El tipo de datos abstracto cumple la siguiente semántica:

- El estado interno del semáforo cuenta cuantos procesos todavía pueden utilizar el recurso. Se puede realizar, p.e., con un número entero que nunca llega a ser negativo.
- Exiten tres operaciones con un semáforo: `init()`, `wait()`, y `signal()` que realizan lo siguiente:

  **`init()`** : Inicializa el semáforo antes de que cualquier proceso ejecute ni una operación `wait()` ni una operación `signal()` al límite de número de procesos que tengan derecho a acceder el recurso. Si se inicializa con 1, se ha contruido un semáforo binario.

  **`wait()`** : Si el estado indica cero, el proceso se queda atrapado en el semáforo hasta que esté despertado por otro proceso. Si el estado indica que un proceso más puede acceder el recurso se decrementa el contador y la operación termina con exito.

  La operación `wait()` tiene que estár implementada como una instrucción atómica. Sin embargo, en muchas implementaciones la operación `wait()` puede ser interrumpida. Normalmente existe una forma de comprobar si la salida del `wait()` es debido a una interrumpción o porque se ha dado acceso al semáforo.

  **`signal()`** : Una vez haber terminado el uso del recurso, el proceso lo señaliza al semáforo. Si queda un proceso bloqueado en el semáforo uno de ellos está despertado, sino se incrementa el contador.

  La operación `signal()` también tiene que estár implementada como instrucción atómica. En algunás implementaciones es posible comprobar si se haya despertado un proceso con exito en caso que uno estaba bloqueado.

Para despertar los procesos existen varias formas que se destinguen en sus grados de justicia:

- FIFO que garantiza que no se produzca inanicón de ningún proceso estando bloqueado en el semáforo
- aleatorio que puede resultar en inanición
- ...

El acceso mutuo a regiones críticas se arregla con un semáforo que permita el acceso a un sólo proceso

```
S.init(1)

P1                         P2
a: loop                        loop
b:   S.wait()                    S.wait()
c:   critical region             critical region
d:   S.signal()                  S.signal()
e:   non-critical region         non-critical region
f: endloop                     endloop
```

Observamos los siguiente detalles:

- Si algún proceso no libera el semáforo, se puede provocar un bloqueo.
- No hace falta que un proceso libere su propio recurso, es decir, la operación `signal()` puede estar ejecutado por otro proceso.
- Con simples semáforos no se puede imponer un orden en los procesos accediendo diferentes recursos.

Si existen en un entorno solamente semáforos binarios, se puede simular un semáforo general usando dos semáforos binarios y un contador, p.e., con las variables `delay`, `mutex` y `count`.

La operación `init()` inicializa el contador al número máximo permitido. El semáforo `mutex` asegura acceso mutuamente exclusivo al contador. El semáforo `delay` atrapa a los procesos que superan el número máximo permitido.

La operación `wait()` se implementa de la siguiente manera:

```
delay.wait()
mutex.wait()
decrement count
if count greater 0 then delay.signal()
mutex.signal()
```

y la operación `signal()` se implementa de la siguiente manera:

```
mutex.wait()
increment count
if count equal 1 then delay.signal()
mutex.signal()
```

Unas principales desventajas de semáforos son:

- no se puede enforzar el uso correcto de los `wait()`s y `signal()`s
- no existe una asociación entre el semáforo y el recurso
- entre `wait()` y `signal()` el usuario puede realizar cualquier operación con el recurso

## 13.2 Regiones críticas

El término abstracto de regiones críticas (Section 9.3) se puede ver realizado directamente en un lenguaje de programación. Así parte de la responsabilidad se ha trasladado desde el programador al compilador.

De algúna manera se identifica que algún bloque de código se debe tratar como región crítica:

```
V is shared variable
region V do
  code of critical region
```

El compilador asegura que la variable `V` tenga un semáforo adjunto que se usa para controlar el acceso exclusivo de un solo proceso a la región crítica. De este modo no hace falta que el programador use directamente las operaciones `wait()` y `signal()` para controlar el acceso con el posible error de olvidarse de algún `signal()`.

Adicionalmente es posible que dentro de la región crítica se llama a otra parte del programa (p.e., un procedimiento o función) que a su vez contenga una región crítica. Si esta región esté controlada por la misma variable `V` el proceso obtiene automáticamente también acceso a dicha región.

Regiones críticas no son lo mismo que los semáforos, porque no se tiene acceso directo a las operaciones `init()`, `wait()` y `signal()`.

## 13.3 Regiones críticas condicionales

En muchas situaciones es conveniente controlar el acceso de varios procesos a una región crítica por una condición. Con las regiones críticas (Section 13.2) simples no se puede realizar tal control. Hace falta otra construccion, por ejemplo:

```
V is shared variable
C is boolean expression
region V when C do
  code of critical region
```

En la programación orientada a objectos (Section 17) las regiones críticas condicionales donde la condición refleja el estado actual de un objeto son muy útiles para controlar el acceso al objeto dependiendo de este estado.

Las regiones críticas condicionales funcionan internamente de la siguiente manera:

1. un proceso que quiere entrar la región crítica espera hasta que tenga permiso
2. una vez obtenido permiso comprueba el estado de la condición, si la condición lo permite entra la región, en caso contrario libera el cerrojo y se pone de nuevo esperando en la cola de acceso

Se implementa una región crítica normalmente con dos colas diferentes. Una cola principal controla los procesos que quieren acceder la región crítica, una cola de eventos controla los procesos que ya han obtenido una vez el cerrojo pero que han encontrado la condición en estado falso. Si un proceso sale de la región crítica todos los procesos que quedan en la cola de eventos pasan de nuevo a la cola principal porque tienen que recomprobar la condición.

Nota que esta técnica puede resultar en muchos comprobaciones de la condición, todos en modo exclusivo, y pueden causar perdidas de eficiencia. En ciertas circumstancias hace falta un control más sofisticado del acceso a la región crítica dando paso directo de un proceso a otro.

## 13.4 Monitores

Todas las estructuras que hemos visto hasta ahora siguen provocando problemas para el programador:

- el control sobre los recursos está distribuido por varios puntos de un programa
- no hay protección de los variables de control que siempre fueron globales

Por eso se usa el concepto de monitores que implementan un nivel aún más alto de abstracción facilitando el acceso a recursos compartidos.

Un monitor es un tipo de datos abstracto que contiene

- un conjunto de procedimientos de control de los cuales solamente un subconjunto es visible desde fuera
- un conjunto de datos privados, es decir, no visibles desde fuera

El acceso al monitor está permito solamente a través de los procedimientos públicos y el compilador garantiza exclusión mutua para todos los procedimientos. La implementación del monitor controla la exclusión mutua con colas de entrada que contengan todos los procesos bloqueados. Pueden existir varias colas y el controlador del monitor elige de cual cola se va a escoger el siguiente proceso para actuar sobre los datos. Un monitor no tiene acceso a variables exteriores con el resultado que su comportamiento no puede depender de ellos.

Una desventaja de los monitores es la exclusividad de su uso, es decir, la concurrencia está limitada si muchos procesos hacen uso del mismo monitor.

Un aspecto que se encuentra en muchas implementaciones de monitores es la sincronización condicional, es decir, mientras un proceso está ejecutando un procedimiento del monitor (con exclusión mutua) puede dar paso a otro proceso liberando el cerrojo. Estas operaciones se suele llamar `wait()` o `delay()`. El proceso que ha liberado el cerrojo se queda bloqueado hasta que otro proceso le despierta de nuevo. Este bloqueo temporal está realizado dentro del monitor.

La técnica permite la sincronización entre procesos porque actuando sobre el mismo recurso los procesos pueden cambiar el estado del recurso y pasar así información de un proceso al otro.

Lenguajes de alto nivel que facilitan la programación concurrente suelen tener monitores implementados dentro del lenguaje (p.e. Java usa el concepto de monitores (Section 18.4.5) para realizar el acceso mutuamente exclusivo a sus objetos).

# 14 Planificador

Proceses pueden estar en varios estados:

Aquí debe venir figura procest.ps

# 15 Problema del productor y consumidor

El problema del productor y consumidor consiste en la situación que de una parte se produce algún producto (datos en nuestro caso) que se posiciona en algún lugar (una cola en nuestro caso) para que sea consumido por otra parte. Como algoritmo obtenemos:

```
producer:                    consumer:
  forever                      forever
    produce(item)                take(item)
    place(item)                  consume(item)
```

Queremos garantizar que el consumidor no coja los datos más rápido que les está produciendo el productor. Más concreto:

1. el productor puede generar sus datos en cualquier momento
2. el consumidor puede coger un dato solamente cuando hay alguno
3. para el intercambio de datos se usa una cola a la cual ambos tienen acceso, asi se garantiza el orden correcto
4. ningún dato no está consumido una vez siendo producido

Si la cola puede crecer a una longitud infinita (siendo el caso cuando el consumidor consume más lento que el productor produzca), basta con la siguiente solución:

```
producer:                    consumer:
  forever                      forever
    produce(item)                itemsReady.wait()
    place(item)                  take(item)
    itemsReady.signal()          consume(item)
```

donde `itemsReady` es un semáforo general que se ha inicializado al principio con el valor `0`.

El algoritmo es correcto que se vee con la siguiente prueba. Asumimos que el consumidor adelanta el productor. Entonces el número de `wait()`s tiene que ser más grande que el número de `signals()`:

```
      #waits > #signals
==>   #signals - #waits < 0
==>   itemsReady < 0
```

y la última línea es una contradicción a la invariante del semáforo.

Si queremos ampliar el problema introduciendo más productores y más consumidores que trabajan todos con la misma cola para asegurar que todos los datos estén consumidos lo más rápido posible por algún consumidor disponible tenemos que proteger el acceso a la cola con un semáforo binario (llamado `mutex` abajo):

```
producer:                   consumer:
  forever                     forever
    produce(item)               itemsReady.wait()
    mutex.wait()                mutex.wait()
    place(item)                 take(item)
    mutex.signal()              mutex.signal()
    itemsReady.signal()         consume(item)
```

Normalmente no se puede permitir que la cola crezca infinitamente, es decir, hay que evitar producción en exceso también. Como posible solución introducimos otro semáforo general (llamado `spacesLeft`) que cuenta cuantos espacios quedan libre en la cola. Se inicializa el semáforo con la longitud máxima permitida de la cola. Un productor queda bloqueado si ya no hay espacio en la cola y un consumidor signala su consumisión.

```
producer:                   consumer:
  forever                     forever
    spacesLeft.wait()           itemsReady.wait()
    produce(item)               mutex.wait()
    mutex.wait()                take(item)
    place(item)                 mutex.signal()
    mutex.signal()              consume(item)
    itemsReady.signal()         spacesLeft.signal()
```

# 16 Bloqueo

Un bloqueo se produce cuando un proceso está esperando a algo que nunca se cumple.

**Ejemplo:**

Cuando dos procesos $P_0$ y $P_1$ quieren tener acceso simultaneamente a dos recursos $r_0$ y $r_1$, es posible que se produzca un bloqueo de ambos procesos. Si $P_0$ aloca con exito $r_1$ y $P_1$ aloca con exito $r_0$, ambos se quedan atrapados intendando tener acceso al otro recurso.

Cuatro condiciones se tienen que cumplir para que sea posible que se produce un bloqueo entre procesos:

1. los procesos tienen que compartir recursos con exclusión mutua
2. los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro
3. los recursos no permiten ser usados por más de un proceso al mismo tiempo
4. existe una cadena circular entre peticiones de procesos y alocación de recursos

Un problema adicional con los bloqueos es que es posible que el programa siga funcionando correcto según la definición, es decir, el resultado obtenido es el resultado deseado, aún unos de sus procesos están bloqueados durante la ejecución.

Existen algunas técnicas que se puede usar para que no se produzcan bloqueos:

## 16.1 Detectar y actuar

Se implementa un proceso adicional que vigila si los demás forman una cadena circular.

Más preciso, se define el grafo de alocación de recursos:

- Los procesos y los recursos representan los nodos de un grafo.
- Se añade cada vez una arista entre un nodo tipo recurso y un nodo tipo proceso cuando el proceso ha obtenido acceso exclusivo al recurso.
- Se añade cada vez una arista entre un nodo tipo recurso y un nodo tipo proceso cuando el proceso está pidiendo acceso exclusivo al recurso.
- Se elimina las aristas entre proceso y recurso y al revés cuando el proceso ya no usa el recurso.

Aquí debe venir figura gra.ps

Cuando se detecta en el grafo resultante un círculo, es decir, cuando ya no forma un grafo acíclico, se ha producido una posible situación de un bloqueo.

Se puede reaccionar en dos maneras si se ha encontrado un ciclo:

- No se da permiso al último proceso de alocar el recurso.
- Sí se da permiso, pero una vez detectado el ciclo se aborta todos/algunos de los procesos involucrados.

Sin embargo, la técnica puede dar como resultado que el programa no avance, es decir, el programa se queda atrapado haciendo trabajo inútil: crear situaciones de bloqueo y abortar procesos continuamente.

## 16.2 Evitar

El sistema no da permiso de acceso a recursos si es posible que el proceso se bloquea en el futuro. Un método es el algoritmo del bancario (Dijkstra) que es un algoritmo centralizado y por eso posiblemente no muy practicable en muchas situaciones.

Se garantiza que todos los procesos actuan de la siguiente manera en dos fases:

1. primero se obtiene todos los cerrojos necesarios para realizar una tarea, eso se realiza solamente si se puede obtener todos a la vez,
2. despues se realiza la tarea durante la cual posiblemente se libra recursos que no son necesarias.

**Ejemplo:**

Asumimos que tengamos 3 procesos que actuan con varios recursos. El sistema dispone de 12 recursos.

| proceso | recursos pedidos | recursos reservados |
|---|---|---|
| A | 4 | 1 |
| B | 6 | 4 |
| C | 8 | 5 |
| suma | 18 | 10 |

es decir, de los 12 recursos disponibles ya 10 están ocupados. La única forma que se puede proceder es dar el aceso a los restantes 2 recursos al proceso B. Cuando B haya terminado va a liberar sus 6 recursos que incluso pueden estar distribuidos entre A y C, así que ambos también pueden realizar su trabajo.

Con un argumento de inducción se verifica fácilmente que nunca se llega a ningún bloqueo.

## 16.3 Prevenir

Se puede prevenir bloqueo siempre cuando se consiga que alguna de las condiciones necesarias para la existencia de un bloqueo no se produce.

1. los procesos tienen que compartir recursos con exclusión mutua:
   - No se de a un proceso directamente acceso exclusivo al recurso, si no se usa otro proceso que realiza el trabajo de todos los demás manejando una cola de tareas (p.e., un demonio para imprimir con su cola de documentos por imprimir).
2. los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro:
   - Se exige que un proceso pide todos los recursos que va a utilizar al comienzo de su trabajo
3. los recursos no permiten ser usados por más de un proceso al mismo tiempo:
   - Se permite que un proceso aborta a otro proceso con el fin de obtener acceso exclusivo al recurso. Hay que tener cuidado de no caer en livelock

4. existe una cadena circular entre peticiones de procesos y alocación de recursos:
   - Se ordena los recursos línealmente y se forza a lo procesos que acceden a los recursos en el orden impuesto. Así es imposible que se produzca un ciclo.

# 17 Programación orientada a objetos

Programar con objetos tiene muchas ventajas que no repetimos aquí. Lo que destaca, a lo mejor, es el uso abstracto de objeto, es decir, se define un objeto por su compartamiento y menos por su estado.

Desde siempre, el concepto de concurrencia fue parte de programación orientada a objetos (ya con Simula67). Ha tenido casi un renacimiento con la aparencia de Java.

Las operaciones con objetos se puede clasificar en cuatro grupos:

- actualizar/modificar el estado actual
- acceptar mensages (de otros objetos)
- mandar mensages (a otros objetos)
- crear/initializar el objeto

Normalmente también se tiene que destruir un objeto al final de su tiempo de vida. Aún eso no se tiene que hacer necesariamente explícitamente.

En un nivel de modelación abstracta se distingue objetos activos y objetos pasivos.

El modelo de objetos activos describe un objeto como una entidad con propia vida que actua cada vez que reciba un mensaje. La actuación puede ser cualquiera de las operaciones mencionadas arriba.

El modelo de objetos pasivos describe un objeto como un conjunto de datos que se modifica bajo control de una administración externa al objeto. Es como si algún interpretador simulase el comportamiento del objeto.

**Ejemplo:**

Dentro de un entorno Java, la máquina virtual ejercita el papel del administrador, tratando todos los objetos del programa como objetos pasivos bajo su control. Sin embargo, al mismo tiempo, cada uno de los objetos puede jugar un papel de un objeto activo que no sabe nada sobre el controlador.

Otra ilustración del concepto sería: un objeto activo está ejecutando su propio hilo, mientras un objeto pasivo se manipula dentro de otro hilo.

¿Porqué se dice mandar mensajes?

Usando objetos en el diseño de programas concurrentes los objetos tendrán también propriedades como

- inmutuabilidad
- posibles cerrojos
- dependencias entre sus estados

# 18 Java

El famoso *hola mundo* se programa en Java así:

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Hello world");
  }
}
```

Antes de explicar rápidamente el lenguaje Java jugamos con un simple ejemplo para motivar el uso de Java en las clases de concurrencia.

## 18.1 Hilos de Java

Hasta ahora hemos hablado siempre de procesos en un contexto abstracto.

Una vez programando en un lenguaje de programación, p.e., C++ o Java, se puede usar sus hilos (“threads”) como instancias de procesos. No obstante, existen más posibilidades de instanciar los procesos.

Un hilo es una secuencia de instrucciones que está controlada por un planificador. El planificador gestiona el tiempo de ejecución del procesador y asigna de alguna manera dicho tiempo a los diferentes hilos actualmente presentes.

Normalmente los hilos de un proceso (en este contexto el proceso es lo que se suele llamar así en el ámbito de sistemas operativos) suelen tener acceso a todos los recursos disponibles al proceso, es decir, actuan sobre una memoria compartida.

Los hilos están en el paquete

`java.lang.thread`

y se puede usar por ejemplo dos hilos para realizar un pequeño pingPONG:

```
Thread PingThread = new Thread();
PingThread.start();
Thread PongThread = new Thread();
PongThread.start();
```

Por defecto, un hilo nuevamente creado y lanzado aún siendo activado así no hace nada. Sin embargo, los hilos se ejecutan durante un tiempo infinito y hay que abortar el programa de forma bruta: control-C en el terminal.

Extendemos la clase y sobre-escribimos el método `run()` para que haga algo útil:

```
public class CD_PingThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("ping ");
    }
  }
}
```

El hilo herede todo de la clase `Thread`, pero sobre-escribe el método `run()`. Hacemos lo mismo para el otro hilo:

```
public class CD_PongThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("PONG ");
    }
  }
}
```

Y reprogramamos el hilo principal:

```
CD_PingThread PingThread=new CD_PingThread();
PingThread.start();
CD_PongThread PongThread=new CD_PongThread();
PongThread.start();
```

**Resultado (esperado):**

- los dos hilos producen cada uno por su parte sus salidas en la pantalla

**Resultado (observado):**

- se ve solamente la salida de un hilo durante cierto tiempo
- parece que la salida dependa cómo el planificador está realizado en el entorno Java

Nuestro objetivo es: la ejecución del pingPONG independientemente del sistema debajo. Intentamos introducir una pausa para "motivar" el planificador para que cambie los hilos:

```
public class CD_PingThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("ping ");
      try {
        sleep(10);
      }
      catch(InterruptedException e) {
        return;
      }
    }
  }
}
```

```
public class CD_PongThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("PONG ");
      try {
        sleep(50);
      }
      catch(InterruptedException e) {
        return;
      }
    }
  }
}
```

**Resultado (observado):**

- se ve un poco más ping que PONG
- incluso si los dos tiempos de espera son iguales no se ve ningún pingPONG perfecto

Existe el método `yield()` (cede) para avisar explícitamente el planificador que debe cambiar los hilos:

```
public class CD_PingThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("ping ");
      yield();
    }
  }
}
```

```
public class CD_PongThread extends Thread {
  public void run() {
    while(true) {
      System.out.print("PONG ");
      yield();
    }
  }
}
```

**Resultado (observado):**

- se ve un ping y un PONG alternativamente, pero de vez en cuando aparecen dos pings o dos PONGs
- parece que el planificador re-seleccione el mismo hilo que ha lanzado el `yield()`

Prácticas: codificar los ejemplos y "jugar con el entorno".

## 18.2 Repaso de Java

### 18.2.1 Clases

Java usa (con la excepción de variables de tipos simples) exclusivamente objetos. Un tal objeto se define como una clase (`class`), y se puede crear varias instancias de objetos de tal clase. Es decir, la clase define el tipo del objeto, y la instancia es una variable que representa un objeto.

Una clase contiene como mucho tres tipos de miembros:

- instancias de objetos (o de tipos simples)
- métodos (funciones)
- otras clases

No existen variables globales y el programa principal no es nada más que un método de una clase.

Los objetos en Java siempre tienen valores conocidos, es decir, los objetos (y también las variables de tipos simples) siempre están inicializados. Si el programa no da una inicialización explícita, Java asigna el valor cero, es decir, `0`, `0.0`, `\u0000`, `false` o `null` dependiendo del tipo de la variable.

Java es muy parecido a C++, aún también existen grandes diferencias.

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Hello world");
  }
}
```

El programa principal se llama `main()` y tiene que ser declarado público y estático. No devuelve ningún valor (por eso se declara como `void`). Los parámetros de la línea de comando se pasa como un arreglo de cadenas de letras (`String`).

Java exige una disciplina estricta con sus tipos, es decir, el compilador controla siempre cuando pueda si las operaciones usadas están permitidas con los tipos involucrados. Si la comprobación no se puede realizar durante el tiempo de compilación, se pospone hasta el tiempo de ejecución, es decir, se pueden provocar excepciones que pueden provocar fallos durante la ejecución.

### 18.2.2 Modificadores de clases

Se puede declarar clases con uno o varios de los siguientes modificadores para especificar ciertas propriedades (no existen en C++):

- `public` la clase es visible desde fuera del fichero
- `abstract` la clase todavía no está completa, es decir, no se puede instanciar objetos antes de que se haya implementado los métodos que faltan en una clase derivada

- `final` no se puede extender la clase
- `strictfp` obliga a la máquina virtual de cumplir el estándar de IEEE para los números flotantes

Casi todos los entornos de desarrollo para Java permiten solamente una clase pública dentro del mismo fichero.

Obviamente una clase no puede ser al mismo tiempo final y abstracto. Tampoco está permitida una clase abstracta con `strictfp`.

### 18.2.3 Comentarios

Existen tres posibilidades de escribir comentarios:

| | |
|---|---|
| `/* ... */` | comentario de bloque |
| `//` | comentario de línea |
| `/** ... */` | comentario de documentación |

### 18.2.4 Tipos simples

| | |
|---|---|
| `boolean` | o bién `true` o bién `false` |
| `char` | 16 bit Unicode letra |
| `byte` | 8 bit número entero con signo |
| `short` | 16 bit número entero con signo |
| `int` | 32 bit número entero con signo |
| `long` | 64 bit número entero con signo |
| `float` | 32 bit número flotante |
| `double` | 64 bit número flotante |

Solo `float` y `double` son igual como en C++. No existen enteros sin signos.

Los tipos simples no son clases, pero existen para todos los tipos simples clases que implementan el comportamiento de ellos. Sólo hace falta escribirles con mayúscula (con la excepción de `Integer`). Las clases para los tipos simples proporcionan también varias constantes para trabajar con los números (p.e., `NEGATIVE_INFINITY` etc.).

### 18.2.5 Modificadores de miembros

Modificadores de acceso:

- `private`: accesible solamente desde la propia clase
- `package`: (o ningún modificador) accesible solamente desde la propia clase o dentro del mismo paquete
- `protected`: accesible solamente desde la propia clase, dentro del mismo paquete, o desde clases derivadas

- `public`: accesible siempre cuando la clase es visible

(En C++, por defecto, los miembros son privados, mientras en Java los miembros son por defecto del paquete.)

Modificadores de miembros siendo instancias de objetos:

- `final`: declara constantes (diferencia a C++ donde se declara constantes con `const`), aún las constantes no se pueden modificar en el transcurso del programa, pueden ser calculadas durante sus construcciónes; variables finales, aún declaradas sin inicialización, tienen que obtener sus valores como muy tarde en la fase de construcción de un objeto de la clase
- `static`: declara miembros de la clase que pertenecen a la clase y no a instancias de objetos, es decir, todos los objetos de la clase acceden la misma cosa
- `transient`: excluye un miembro del proceso de conversión en un flujo de bytes si el objeto se salva al disco o se transmite por una conexión (no hay en C++)
- `volatile`: ordena la máquina virtual de Java de no usar ningún tipo de cache para el miembro, así es más probable (aún no garantizado) que varios hilos ven el mismo valor de una variable; declarando variables del tipo `long` o `double` como `volatile` asegura que las operaciones básicas son atómicas

Modificadores de miembros siendo métodos:

- `abstract`: el método todavía no está completo, es decir, no se puede instanciar objetos antes de que se haya implementado el método en una clase derivada (parecido a los métodos puros de C++)
- `static`: el método pertenece a la clase y no a un objeto de la clase, un método estático puede acceder solamente miembros estáticos
- `final`: no se puede sobre-escribir el método en una clase derivada (no hay en C++)
- `synchronized`: el método pertenece a una región crítica del objeto (no hay en C++)
- `native`: propone una interfaz para llamar a métodos escritos en otros lenguajes, su uso depende de la implementación de la máquina virtual de Java (no hay en C++, ahí se realiza durante el linkage)
- `strictfp`: obliga a la máquina virtual de cumplir el estándar de IEEE para los números flotantes (no hay en C++, ahí depende de las opciones del compilador)

Un método abstracto no puede ser al mismo tiempo ni final, ni estático, ni sincronizado, ni nativo, ni estricto.

Un método nativo no puede ser al mismo tiempo ni abstracto ni estricto.

Nota que el uso de `final` y `private` puede mejorar las posibilidades de optimización del compilador, es decir, su uso resulta en programas más eficientes.

### 18.2.6 Estructuras de control

Las estructuras de control son iguales a las de C++:

- `if (cond) then expr;`
- `if (cond) then expr else expr;`
- `while (cond) expr;`
- `do expr; while (cond);`
- `for(expr; expr; expr) expr;`
- `switch (expr) { case const: ... default: }`

Igual como en C++ se puede declarar una variable en la expresión condicional o dentro de la expresión de inicio del bucle `for`.

Adicionalmente Java proporciona `break` con una marca que se puede usar para salir en un salto de varios bucles intercalados.

```
mark:
  while(...) {
    for(...) {
      break mark;
    }
  }
```

Igual existe un `continue` con marca que permite saltar al principio de un bucle más alla del actual.

No existe el `goto`, su uso habitual en C++ se puede emular (mejor) con los `break`s y `continue`s y con las secuencias `try-catch-finally`.

### 18.2.7 Operadores

Java usa los mismos operadores que C++ con las siguientes excepciones:

- existe adicionalmente `>>>` como desplazamiento a la derecha llenando con ceros a la izquierda
- existe el `instanceof` para comparar tipos
- los operadores de C++ relacionados a punteros no existen
- no existe el `delete` de C++
- no existe el `sizeof` de C++

La prioridad y la asociatividad son las mismas que en C++. Hay pequeñas diferencias entre Java y C++ si ciertos símbolos están tratados como operadores o no (p.e., los `[]`). Además Java no proporciona la posibilidad de sobre-cargar operadores.

### 18.2.8 Palabras reservadas

Las siguientes palabras están reservadas en Java:

```
abstract  default  if          private       this
boolean   do       implements  protected     throw
break     double   import      public        throws
byte      else     instanceof  return        transient
case      extends  int         short         try
catch     final    interface   static        void
char      finally  long        strictfp      volatile
class     float    native      super         while
const     for      new         switch
continue  goto     package     synchronized
```

Además las palabras `null`, `false` y `true` que sirven como constantes no se pueden usar como nombres. Aún `goto` y `const` aparecen en la lista arriba, no se usan en el lenguaje.

### 18.2.9 Objetos y referencias a objetos

No se puede declarar instancias de clases usando el nombre de la clase y un nombre para el objeto (como se hace en C++). La declaración

```
ClassName ObjectName
```

crea solamente una referencia a un objeto de dicho tipo. Para crear un objeto dinámico en el montón se usa el operador `new` con el constructor del objeto deseado. El operador devuelve una referencia al objeto creado.

```
ClassName ObjectReference = new ClassName(...)
```

Sólo si una clase no contiene ningún constructor Java propone un constructor por defecto que tiene el mismo modificador de acceso que la clase. Constructores pueden lanzar excepciones como cualquier otro método.

Para facilitar la construcción de objetos aún más, es posible de usar bloques de código sin que pertenezcan a constructores. Esos bloques están prepuestos (en su orden de aparencia) delante de los códigos de todos los constructores.

El mismo mecanismo se puede usar para inicializar miembros estáticos poniendo un `static` delante del bloque de código. Inicializaciones estáticas no pueden lanzar excepciones.

```
class ... {
  ...
  static int[] powertwo=new int[10];
  static {
    powertwo[0]=1;
    for(int i=1; i<powertwo.length; i++)
      powertwo[i]=powertwo[i-1]<<1;
  }
  ...
}
```

Si una clase, p.e., `X`, construye un miembro estático de otra clase, p.e., `Y`, y al revés, el bloque de inicialización de `X` está ejecutado solamente hasta la aparencia de `Y` cuyos bloques de inicialización recurren al `X` construido a medias. Nota que todas las variables en Java siempre están en cero si todavía no están inicializadas explícitamente.

No existe un operador para eliminar objetos del montón, eso es tarea del recolector de memoria incorporado en Java (diferencia a C++ donde se tiene que librar memoria con `delete` explícitamente).

Para dar pistas de ayuda al recolector se puede asignar `null` a una referencia indicando al recolector que no se va a referenciar dicho objeto nunca jamás.

Las referencias que todavía no acceden a ningún objeto tienen el valor `null`.

Está permitida la conversión explícita de un tipo a otro mediante el "cast" con todas sus posibles consecuencias.

El "cast" es importante especialmente en su variante del "downcast", es decir, cuando se sabe que algún objeto es de cierto tipo derivado pero se tiene solamente una referencia a una de sus super-clases.

Se puede comprobar el tipo actual de una referencia con el operador `instanceof`.

```
if( refX instanceof refY ) { ... }
```

### 18.2.10 Parámetros

Se puede pasar objetos como parámetros a métodos.

La lista de parámetros junto con el nombre del método compone la signatura del método. Pueden existir varias funciones con el mismo nombre, siempre cuando se distinguen en sus signaturas. La técnica se llama sobrecarga de métodos.

La lista de parámetros siempre es fija, no existe el concepto de listas de parámetros variables de C.

Java pasa parámetros exclusivamente con sus valores. Eso significa en caso de objetos que siempre se pasa una referencia al objeto con la consecuencia de que el método llamado puede modificar el objeto.

Para evitar posibles modificaciones de un parámetro se puede declarar el parámetro como `final`.

Entonces, no se puede cambiar los valores de variables de tipos simples llamando a métodes y pasarles como parámetros variables de tipos simples.

### 18.2.11 Valores de retorno

Un método termina su ejecución en tres ocaciones:

- se ha llegado al final de su código
- se ha encontrado una sentencia `return`
- se ha producido una excepción no tratada en el mismo método

Un `return` con parámetro cuyo tipo tiene que coincidir con el tipo del método devuelve una referencia a una variable de dicho tipo (o el valor en caso de tipos simples).

### 18.2.12 Arreglos

Los arreglos se declaran solamente con su límite súperior dado que el límite ínferior siempre es cero (`0`).

El código

```
int[] vector = new vector[15]
```

crea un vector de números enteros de longitud 15.

Java comprueba si los accesos a arreglos con índices quedan dentro de los límites permitidos (diferencia a C++ donde no hay una comprobación). Si se detecta un acceso fuera de los límites se produce una excepción `IndexOutOfBoundsException`.

Arreglos son objetos implícitos que siempre conocen sus propias longitudes (`values.length`) (diferencia a C++ donde un arreglo es nada más que un puntero) y que se comportan como clases finales.

No se puede declarar los elementos de un arreglo como constantes (como es posible en C++).

### 18.2.13 `this` and `super`

Cada objeto tiene por defecto una referencia llamada `this` que proporciona acceso al propio objeto (diferencia a C++ donde `this` es un puntero).

Obviamente, la referencia `this` no existe en métodos estáticos.

Cada objeto (menos la clase `object`) tiene una referencia a su clase súperior llamada `super` (diferencia a C++ donde no existe, se tiene acceso a las clases superiores por otros medios).

`this` y `super` se pueden usar especialmente para acceder a variables y métodos que están escondidos por nombres locales.

Para facilitar las definiciones de constructores, un constructor puede llamar en su primer sentencia o bién a otro constructor con `this(...)` o bién a un constructor de su super-clase con `super(...)` (ambos no exiten en C++). El constructor de la super-clase sin parámetros está llamado en todos los casos al final de la posible cadena de llamadas a constructores `this()` en caso que no haya una llamada explícita.

La construcción de objetos sigue siempre el siguiente orden:

- construcción de la super-clase, nota que no se llama ningún constructor por defecto que no sea el constructor sin parámetros
- ejecución de todos los bloques de inicialización
- ejecución del código del constructor

### 18.2.14 Extender clases

Se puede crear nuevas clases extender clases ya existentes (en caso que no sean finales). Las nuevas clases se suelen llamar sub-clases o clases extendidas.

Una sub-clase heredere todas las propriedades de la clase súperior, aún se tiene solamente acceso directo a las partes de la super-clase declaradas por lo menos `protected`.

No se puede extender al mismo tiempo de más de una clase súperior (diferencia a C++ donde se puede derivar de más de una clase).

Se puede sobre-escribir métodos de la super-clase. Si se ha sobre-escrito una cierta función, automáticamente todas las funciones con el mismo nombre aún otras signaturas ya no están accesibles de modo directo.

Si se quiere ejecutar dentro de un método sobre-escrito el código de la super-clase, se puede acceder el método original con la referencia `super`.

Se puede como mucho extender la accesibilidad de métodos sobre-escritos. Se puede cambiar los modificadores del método. También se puede cambiar si los parámetros del método son finales o no, es decir, `final` no forma parte de la signatura.

Los tipos de las excepciones que lanza un método sobre-escrito tienen que ser un subconjunto de los tipos de las excepciones que lanza el método de la super-clase. Dicho subconjunto puede ser el conjunto vacio.

Se se llama a un método dentro de una jerarquía de clases, se ejecuta siempre la versión del método que corresponde al objeto creado (y no necesariamente al tipo de referencia dado) respetando su accesibilidad. Está técnica se llama polimorfismo.

### 18.2.15 Clases dentro de clases

Se puede declarar dentro de clases otras clases. Sin embargo, dichas clases no pueden tener miembros estáticos no-finales.

Todos los miembros de la clase alrededor están visibles desde la clase ínterior (diferencia a C++ donde hay que declarar la clase ínterior como `friend` para obtener dicho efecto).

Extender clases ínteriores se hace igual como clases normales; solamente hay que tener en cuenta que para una clase ínterior siempre hace falta la existencia de un objeto de su clase alrededor antes de que se pueda construir, es decir, tiene que ser claro de dónde viene su `super`.

### 18.2.16 Clases locales

Dentro de cada bloque de código se puede declarar clases locales que son visibles solamente dentro de dicho bloque.

### 18.2.17 La clase `Object`

Todos los objetos de Java son extensiones de la clase `Object`. Los métodos publicos y protegidos de esta clase son

- `public boolean equals(Object obj)`
  compara si dos objetos son iguales, por defecto un objeto es igual solamente a si mismo
- `public int hashCode()` devuelve (con alta probabilidad) un valor distinto para cada objeto
- `protected Object clone() throws CloneNotSuportedException` devuelve una copia binaria del objeto (¡incluyendo sus referencias!)

- `public final Class getClass()` devuelve el objeto del tipo `Class` que representa dicha clase durante la ejecución
- `protected void finalize() throws Throwable` se usa para finalizar el objeto, es decir, avisar al administrador de la memoria que ya no se usa dicho objeto
- `public String toString()` devuelvo una cadena describiendo el objeto

Clases derivadas deben sobre-ecribir los métodos adecuadamente, p.e., el método `equals` si se requiere una comparación binaria.

### 18.2.18 Clonar objetos

Lo vemos cuando nos haga falta.

### 18.2.19 Interfaces

Usando `interface` en vez de `class` se define una interfaz a una clase sin especificar el código de los métodos.

Una interfaz no es nada más que una especificación cómo algo debe ser implementado para que se pueda usar en otro código.

Una interfaz no puede tener declaraciones de objetos que no son ni constantes (`final`) ni estáticos (`static`). En otras palabras, todas las declaraciones de objetos automáticamente son finales y estáticos, aún no se ha escrito explícitamente.

Igual como clases, interfaces pueden incorporar otras clases o interfaces. También se pueden extender interfaces. Nota que es posible extender una interfaz a base de más de una interfaz:

```
interface ThisOne extends ThatOne, OtherOne { ... }
```

Todos los métodos de una interfaz son implícitamente públicos y abstractos, aún no se ha escrito ni `public` ni `abstract` explícitamente (y eso es la convención).

Los demás modificadores no están permitidos para métodos en interfaces.

Para generar un programa todas las interfaces usadas tienen que tener sus clases que las implementan.

Una clase puede implementar varias interfaces al mismo tiempo (aún una clase puede extender como mucho una clase). Se identifican las interfaces implementadas con `implements` después de una posible extensión (`extends`) de la clase.

**Ejemplo:**

```
public interface Comparable {
  int compareTo(Object o);
}


class Something extends Anything implements Comparable {
  ...
  public int compareTo(Object o) {
```

```
      // cast to get a correct object
      // may throw exception ClassCastException
    Something s = (Something)o;
    ... // code to compare to somethings
  }
```

Interfaces se comportan como clases totalmente abstractas, es decir, que no tienen ni miembros no-estáticos, nada diferente a público, y ningún código no-estático.

### 18.2.20 Clases y paquetes

Java viene con una amplia gama de clases y paquetes predefinidos, p.e., `AWT`, `Swing`. Aún no están disponibles siempre en todas las plataformas en sus últimas versiones y eso puede resultar en código no portable.

Java proporciona de la clase `String` (cadenas) con muchos métodos ya implementados. Si se requiere muchas operaciones de cadenas que modifican el contenido de la cadena, mejor usar la clase `StringBuffer`.

### 18.2.21 Excepciones

Para facilitar la programación de casos excepcionales Java usa el concepto de lanzar excepciones.

Una excepción es una clase predefinida y se accede con la sentencia

```
try { ... }
catch (SomeExceptionObject e) { ... }
catch (AnotherExceptionObject e) { ... }
finally { ... }
```

- El bloque `try` contiene el código normal por ejecutar.
- Un bloque `catch(ExceptionObject)` contiene el código excepcional por ejecutar en caso que durante la ejecución del código normal (que contiene el bloque `try`) se produce la excepción del tipo adecuado. Pueden existir más de un (o ningún) bloque `catch` para reaccionar directamente a más de un (ningún) tipo de excepción. Hay que tener cuidado en ordenar las excepciones correctamente, es decir, las más específicas antes de las más generales.
- El bloque `finally` se ejecuta siempre una vez haber terminado o bién el bloque `try` o bién un bloque `catch` o bién una excepción no tratada o bién antes de seguir un `break`, un `continue` o un `return` hacía fuera de la `try-catch-finally`-sentencia.

Normalmente se extiende la clase `Exception` para implementar propias clases de excepciones, aún también se puede derivar directamente de la clase `Throwable` que es la super-clase (interfaz) de `Exception` o de la clase `RuntimeException`.

```
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
}
```

Entonces, una excepción no es nada más que un objeto que se crea en caso de aparencia del caso excepcional. La clase principal de una excepción es la interfaz `Throwable` que incluye un `String` para mostrar una línea de error leíble.

Para que un método pueda lanzar excepciones con las `try-catch-finally`-sentencias, es imprecindible declarar las excepciones posibles antes del bloque de código del método con `throws ....`

```
public void myfunc(int arg) throws MyException { ...  }
```

Durante la ejecución de un programa se propagan las excepciones desde su punto de aparición subiendo las invocaciones de los métodos hasta que se haya encontrado un bloque `catch` que se ocupa en tratar la excepción. En el caso que no haya ningún bloque responsable, la excepción está tratada por la máquina virtual con el posible resultado de abortar el programa.

Se puede lanzar excepciones directamente con la palabra `throw` y la creación de un nuevo objeto de excepción, p.e.,

```
throw new MyException("eso es una excepcion");
```

También constructores pueden lanzar excepciones que tienen que ser tratados en los métodos que usan dichos objetos construidos.

Además de las excepciones así declaradas existen siempre excepciones que pueden ocurrir en qualquier momento de la ejecución del programa, p.e., `RuntimeException` o `Error` o `IndexOutOfBoundException`. La ocurrencia de dichas excepciones refleja normalmente un flujo de control erróneo del programa que se debe corrigir antes de distribuir el programa a posibles usuario.

Se usa excepciones solamente para casos excepcionales, es decir, si pasa algo que no se haya esperado.

### 18.2.22 Paquetes

Siempre existe la posibilidad que diferentes fuentes usan el mismo nombre para clases. Para producir nombres únicos se ha introducido el concepto de paquetes. El nombre del paquete sirve como prefijo del nombre de la clase con la consequencia que siempre cuando se diferencian los nombres de los paquetes también se diferencian los nombres de las clases.

Por convención se usa como prefijo el dominio en el internet en orden reverso para los paquetes. Hay que tener cuidado en distinguir los puntos en el nombre del paquete con los puntos que separan los miembros de una clase.

La pertenencia de una clase a un paquete se indica en la primera sentencia de un fichero fuente con

```
package Pack.Name;
```

### 18.2.23 Conversión a flujos de bytes

Lo vemos cuando nos haga falta.

## 18.3 Reflexión

Java proporciona para cada clase un objeto de tipo `Class` que se puede usar para obtener información sobre la propia clase y todos sus miembros.

Así por ejemplo se puede averigar todos sus métodos y modificadores, cual es su clase súperior y mucho más.

Vemos más cuando nos haga falta.

## 18.4 Hilos

Se usan los hilos para ejecutar varias secuencias de instrucciones de modo quasi-paralelo.

### 18.4.1 Crear un hilo

Se crea un hilo con

```
Thread worker = new Thread()
```

Después se inicializa el hilo y se define su comportamiento.

Se lanza el hilo con

```
worker.start()
```

Aún en esta versión simple no hace nada. Hace falta sobre-escribir el método `run()` especificando algún código útil.

Ya vimos el uso de los hilos en la introducción a Java con el simple ejemplo del `pingPONG`.

### 18.4.2 La interfaz `Runnable`

A veces no es conveniente extender la clase `Thread` porque se pierde la posibilidad de extender otro objeto. Es una de las razones por que existe la interfaz `Runnable` que declara nada más que el método `public void run()` y que se puede usar fácilmente para crear hilos trabajadores.

```
class RunPingPONG implements Runnable {
  private String word;
  private int delay;

  RunPingPONG(String whatToSay, int delayTime) {
    word =whatToSay;
    delay=delayTime;
  }

  public void run() {
    try {
      for(;;) {
        System.out.print(word+" ");
        Thread.sleep(delay);
      }
```

```
    }
    catch(InterruptedException e) {
      return;
    }
  }

  public static void main(String[] args) {
    Runnable ping = new RunPingPONG("ping", 40);
    Runnable PONG = new RunPingPONG("PONG", 50);
    new Thread(ping).start();
    new Thread(PONG).start();
  }
}
```

Existen cuatro constructores para crear hilos usando la interfaz `Runnable`.

- `public Thread(Runnable target)`
  así lo usamos en el ejemplo arriba, se pasa solamente la implementación de la interfaz `Runnable`
- `public Thread(Runnable target, String name)`
  se pasa adicionalmente un nombre para el hilo
- `public Thread(ThreadGroup group, Runnable target)`
  construye un hilo dentro de un grupo de hilos
- `public Thread(ThreadGroup group, Runnable target, String name)`
  construye un hilo con nombre dentro de un grupo de hilos

La interfaz `Runnable` exige solamente el método `run()`, sin embargo, normalmente se implementa más métodos para crear un servicio complete que este hilo debe cumplir.

Aún no hemos guardado las referencias de los hilos en unas variables, los hilos no caen en las manos del recolector de memoria: siempre se mantiene una referencia al hilo en su grupo al cual pertenece.

El método `run()` es público y en muchos casos implementando algún tipo de servicio no se quiere dar permiso a otros ejecutar directamente el método `run()`. Para evitar eso se puede recurrir a la siguiente construcción:

```
class Service {
  private Queue requests = new Queue();
  public Service() {
    Runnable service = new Runnable() {
      public void run() {
        for(;;) realService((Job)requests.take());
      }
    };
    new Thread(service).start();
  }
  public void AddJob(Job job) {
    requests.add(job);
  }
```

```
  private void realService(Job job) {
    // do the real work
  }
}
```

Crear el servicio con `Service()` lanza un nuevo hilo que actua sobre una cola para realizar su trabajo con cada tarea que encuentra ahí. El trabajo por hacer se encuentra en el método privado `realService()`. Una nueva tarea se puede añadir a la cola con `AddJob(...)`.

**Nota:** la construcción arriba usa el concepto de clases anónimos de Java, es decir, sabiendo que no se va a usar la clase en otro sitio menos en su punto de construcción, se declara directamente donde se usa.

### 18.4.3 Sincronización

Es posible en Java forzar la ejecución de código en un bloque en modo sincronizado, es decir, como mucho un hilo puede ejecutar algún código dentro de dicho bloque al mismo tiempo.

```
synchronized (obj) { ... }
```

La expresión entre paréntesis `obj` tiene que evaluar a una referencia a un objeto o a un arreglo.

Declarando un método con el modificador `synchronized` garantiza que dicho método se ejecuta ininterrumpidamente por un sólo hilo. La máquina virtual instala un cerrojo (mejor decir, un monitor) que se cierra de forma atómica antes de entrar en la región crítica y que se abre antes de salir.

Declarar un método como

```
synchronized void f() { ... }
```

es equivalente a usar un bloque sincronizado en su interior:

```
void f() { synchronized(this) { ... } }
```

Los monitor permite que el mismo hilo puede acceder otros métodos o bloques sincronizados del mismo objeto sin problema. Se libera el objeto sea cual sea el modo de terminar el método.

Constructores no se pueden declarar `synchronized`.

No hace falta de mantener el modo sincronizado sobre-escribiendo métodos síncronos mientras se extiende una clase. Sin embargo, una llamada al método de la clase súperior (con `super.`) sigue funcionando de modo síncrono.

Métodos estáticos también pueden ser declarados `synchronized` garantizando su ejecución mutuo exclusivo entre varios hilos.

En ciertos casos se tiene que proteger el acceso a miembros estáticos con un cerrojo. Para conseguir eso es posible de sincronizar con un cerrojo de la clase, por ejemplo:

```
class MyClass {
  static private int nextID;
  ...
  MyClass() {
    synchronized(MyClass.class) {
      idNum=nextID++;
    }
```

```
    }
    ...
}
```

**Nota:** declarar un bloque o un método como síncrono solo prevee que ningún otro hilo pueda ejecutar al mismo tiempo dicha región crítica, sin embargo, cualquier otro código asíncrono puede ser ejecutado mientras tanto y su acceso a variables críticas puede dar como resultado fallos en el programa.

### 18.4.4 Objetos síncronos

Se obtiene objetos totalmente sincronizados siguiendo las reglas:

- todos los métodos son `synchronized`,
- no hay miembros/atributos públicos,
- todos los métodos son `final`,
- se inicializa siempre todo bién,
- el estado del objeto se mantiene siempre consistente incluyendo los casos de excepciones.

### 18.4.5 Los monitores de Java

Aquí debe venir figura javamon.ps

## 18.5 Java y seguridad

Muchas veces se oye que Java es un lenguaje seguro porque es tan estricto con sus tipos y la máquina virtual de Java puede prohibir ciertas acciones (como, p.e., escribir al disco o acceder ciertos recursos).

Sin embargo, hay que tener en cuenta que Java no es más seguro que la implementación de la máquina virtual.

## 18.6 Atomicidad en Java

Solo asignaciónes a variables de tipos simples son atómicas (long y double no son simples en este contexto, hay que declararles `volatile` para obtener acceso atómico).

# 19 Concurrencia en memoria distribuida

## 19.1 Paso de mensajes

Un proceso manda un mensaje que es recibido por otro proceso que suele esperar dicho mensaje. El paso de mensajes es imprecindible en sistemas distribuidas dado que en este caso no existen recursos directamente compartidos para intercambiar información entre los procesos. Sin embargo, también si se trabaja con un solo procesador pasar mensajes entre procesos es un buén método de sincronizar procesos o trabajos, respectivamente. Existen muchas variantes de implementaciones de paso de mensajes. Destacamos unas características.

### 19.1.1 Tipos de sincronización

El paso de mensajes puede ser síncrono o asíncrono depende de lo que hace el remitente antes de seguir procesando, más concreto:

- el remitente puede esperar hasta que se haya ejecutado la recepción correspondiente al otro lado; es el método del rendezvous simple o de la comunicación síncrona
- el remitente puede seguir procesando sin esperar al recipiente; es el método de la comunicacón asíncrona
- el remitente puede espear hasta que el recipiente haya contestado al mensaje recibido; es el método del rendezvous extendido o de la involucración remota

Bajo ciertas circunstancias los remitentes y los recipientes pueden implementar una espera finita para no quedar bloqueado infinitamente al no llegar información necesaria del otro lado.

Sobre todo por razones de eficiencia es conveniente de distinguir entre mensajes locales y mensajes a procesadores remotos.

### 19.1.2 Identificación del otro lado

Se puede distinguir varias posibilidades en cómo dos procesos mandan respectivamente reciben sus mensajes:

- se usa nombres únicos para identificar tanto el remitente como el recipiente
  entonces ambos partes tienen que especificar exactamente con que proceso quieren comunicarse
- solo el remitente especifica el destino, al recipiente no le importe quién ha mandado el mensaje (sistema cliente/servidor)
- a ninguno de las dos partes interesa cual será el proceso al otro lado, el remitente manda su mensaje a un buzón de mensajes y el recipiente inspecciona su buzón de mensajes

### 19.1.3 Prioridades

Para el paso de mensajes su usa muchas veces el concepto de un canal entre el remitente y el recipiente o también entre los buzones de mensajes y sus lectores.

Los canales pueden ser capaces de distinguir entre mensajes de diferentes prioridades. Cuando llega un mensaje de alta prioridad, este se adelanta a todos los mensajes que todavía no se haya traspasado al recipiente (p.e. “out-of-band” mensajes en el protocolo `ftp`).

# 20 Terminación de programas

¿Cuáles pueden ser las causas por qué se termina o no se termina un programa?

**terminar con éxito**

**terminar con excepción**

**terminar con suicidio**

**terminar por assesinato**

**terminar por redundancia**

**terminar nunca a propósito**

**terminar nunca por fallo**

Un programa secuencial termina cuando su ha ejecutado su última instrucción. El sistema operativa suele saber cuando eso ocurre.

Sin embargo puede ser difícil detectar cuando un programa concurrente ha terminado, sobre todo cuando también el sistema operativo esté distribuido.

Un programa concurrente termina cuando todos sus partes secuenciales hayan terminado.

Si el sistema distribuido contiene un procesador central que siempre está monitorizando los demás, se puede implementar la terminación igual como en un sistema secuencial.

Si el sistema no dispone de tal procesador central es más difícil porque no se puede observar fácilmente el estado exacto del sistema dado que sobre todos los canales de comunicación se resisten a inspección y pueden contener todavía mensajes no recibidas.

## 20.1 Detección de terminación

Asumimos el siguiente modelo del sistema distribuido:

- El sistema es fiable, es decir, ni los procesos/procesadores no el sistema de comunicación provocan fallos.
- Los procesos que están conectados usan un canal bidireccional para intercambiar mensajes.
- Existe un único proceso que inicia la computación; todos los demás procesos están iniciados por un proceso ya trabajando.

Para detectar la terminación de todos los procesos acordamos lo siguiente:

- se manda mensajes para realizar las tareas del programa

- si un proceso recibe un mensaje lo tiene que procesar
- se manda señales para realizar la detección de terminación
- cuando un proceso se ha decido terminar (sea la razón que sea) no manda más mensajes a los demás, sin embargo, si recibe de nuevo un mensaje reanuda el trabajo
- los canales para los mensajes y los señales existen siempre en pares y los canales de los señales siempre funcionan independiente del estado del proceso o del estado del canal de mensajes

Un ejemplo para la detección de terminación es el algoritmo de Dijkstra-Scholten.

Asumimos primero que el grafo de los procesos (es decir, el grafo que se establece por los intercambios de mensajes entre los procesos) forme un árbol. Esta situación no es tan raro considerando los muchos problemas que se puede solucionar con estrategias de divide-y-vencerás.

La detección de terminación resulta fácil: una hija en el arbol manda y su madre que ha terminado cuando haya recibido el mismo mensaje de todas sus hijas y cuando se ha decido terminar también.

El programa termina cuando la raíz del árbol ha terminado, es decir, cuando ha recibido todas las señales de terminación de todas sus hijas y no queda nada más por hacer. La raíz propaga la decisión que todos pueden terminar definitivamente a lo largo del árbol.

Ampliamos el algoritmo para que funcione también con grafos acíclicos. Añadimos a cada arista un campo “déficit” que se aumenta siempre que se haya pasado un mensaje entre los procesos a ambos lados de la arista. Cuando desea terminar un proceso manda por todas sus aristas entrantes tantas señales como indica el valor “déficit” disminuendo así el campo. Un proceso puede terminar cuando desea terminar y todos sus aristas salientes tengan déficit cero.

El algoritmo de Dijkstra-Scholten desarrollado hasta ahora obviamente no funciona para grafos que contienen cíclos. Sin embargo, se puede usar el siguiente truco:

Siempre cuando un proceso es iniciado por primera vez, el correspondiente mensaje causa una arista nueva en el grafo que es la primera que conecta a dicho proceso. Si marcamos estas aristas especialmente, se observa que forman un árbol abarcador (“spanning tree”) del grafo.

Aquí debe venir figura arbolabar.ps

El algoritmo de determinación de terminación procede entonces como sigue:

- cuando un proceso decide terminar, manda señales según los valores déficit de todas sus arista entrantes menos de las aristas que forman parte del árbol abarcador.
- una vez obtenido todos los déficits (menos los del árbol) igual a cero, se procede igual como en el caso del árbol sencillo.

# 21 Glosario

**critical region** (*región cítica)*

**deadlock** (*interbloqueo)*

**liveness** (*vivacidad)*

**mutual exclusion** (*exclusión mutua)*

**process** (*proceso)*

**semaphore** (*semáforo)*

**scheduler** (*planificador)*

**spanning tree** (*árbol abarcador)*

**starvation** (*inanición)*

**thread** (*hilo)*

**trade-off** (*concesiones mutuas)*

# 22 Bibliografía

1. G.R. Andrews, Concurrent Programming: Principles and Practice, Benjamin/Cummings, 1991.

2. K. Arnold et.al., The Java Programming Language, Addison-Wesley, 3rd Edition, 2000.

3. J.C. Baeten and W.P. Wiejland, Process Algebra, Cambridge University Press 1990.

4. M. Ben-Ari, Principles of Concurrent and Distributed Programming, Prentice-Hall 1990.

5. A. Burns and G.Davies, Concurrent Programming, Addison-Wesley 1993.

6. C. Fencott, Formal Methods for Concurrency, Thomson Computer Press 1996.

7. C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall 1985.

8. Doug Lea, Concurrent Programming in Java, Addison-Wesley, 1997. (Programación Concurrente en Java, Addison-Wesley, 2001.)

9. R. Milner, Concurrency and Communication, Prentice-Hall 1989.

10. R. Milner, Semantics of Concurrent Processes, in J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, Elsevier and MIT Press, 1990.

11. A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall 1997.