# 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
formella@ei.uvigo.es

13 de junio de 2005

# Índice

# 1. Curso

| | |
|---|---|
| **Teoría:** | los lunes, 18-20 horas, Aula 3.2 |
| **Prácticas:** | 3 grupos, los lunes 16-18 horas y los martes 12-14 y 18-20 horas, Laboratorios en el sótano |
| **Asignaturas vecinas:** | todo sobre Programación, Sistemas Operativos, Procesamiento Paralelo, Redes, Sistemas en Tiempo Real, Diseño de Software |
| **Prerrequisitos:** | Java o C/C++, programación secuencial |
| **Evaluación:** | 80 % con un examen escrito al final del curso, teoría y práctica juntos<br>20 % por trabajos realizados durante las prácticas<br>se puede obtener 25 % de los puntos del examen final con trabajos voluntarios durante las prácticas |
| **Créditos:** | 6 (3 teoría, 3 prácticas) |
| **Literatura:** | mira Bibliografía |

# 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. ¿De qué se trata?

Una posible distingción según mi opinión es:

- la programación concurrente se dedica más a *desarrollar* y *aplicar* conceptos para el uso de recursos en paralelo (desde el punto de vista de varios actores)
- la programción en paralelo se dedica más a *solucionar* y *analizar* problemas bajo el concepto del uso de recursos en paralelo (desde el punto de vista de un sólo actor)

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, tienen que intercambiar la información mediante de transmisión de mensajes.

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

## 4.2. Ejemplo

Sumamos los siguientes números:

| | | | |
|---|---|---|---|
| 3482 0984 | 8473 8093 | 3746 6112 | 4958 6432 |
| 9923 7463 | 4398 7329 | 8746 0302 | 9823 4326 |
| 9821 3234 | 8464 5643 | 3745 2854 | 7734 6511 |
| 6534 7732 | 2907 0238 | 2985 5328 | 7334 6532 |
| 3982 6452 | 4328 9231 | 8439 4431 | 8374 4721 |
| 3274 8549 | 3278 8192 | 7843 1723 | 7364 1323 |
| 8329 0123 | 1212 8322 | 4133 7742 | 1232 9234 |
| 6434 6012 | 3823 7213 | 7438 7439 | 3284 2328 |

El resultado:

| | | | |
|---|---|---|---|
| 3482 0984 | 8473 8093 | 3746 6112 | 4958 6432 |
| 9923 7463 | 4398 7329 | 8746 0302 | 9823 4326 |
| 9821 3234 | 8464 5643 | 3745 2854 | 7734 6511 |
| 6534 7732 | 2907 0238 | 2985 5328 | 7334 6532 |
| 3982 6452 | 4328 9231 | 8439 4431 | 8374 4721 |
| 3274 8549 | 3278 8192 | 7843 1723 | 7364 1323 |
| 8329 0123 | 1212 8322 | 4133 7742 | 1232 9234 |
| 6434 6012 | 3823 7213 | 7438 7439 | 3284 2328 |
| 5 1783 0549 | 3 6888 4261 | 4 7078 5931 | 5 0107 1407 |
| | | | 18 5857 2148 |

¿Con qué problemas nos hemos encontrado?:

- selección del algoritmo

- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- fiabilidad de los componentes
- fiabilidad de la comunicación
- detección de la terminación

# 5. Procesos

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

- una secuencia de instrucciones o sentencias que
- se ejecutan 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 que se distinguen en varios aspectos (por ejemplo, en el acceso a los recursos, en la vista de memoria, en la priorisación etc.). En nuestro contexto no vamos a diferenciar 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 (sobre todo aprovechan de una memoria común) 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). El cambio de contexto de un hilo al siguiente dentro del procesador se realiza rápidamente.

Un programa secuencial consiste en un sólo proceso.

En un programa concurrente trabaja un conjunto de procesos en paralelo los cuales cooperan para resolver un problema o realizar una tarea.

Dichos procesos pueden actuar en hardware diferente, es decir, en un ordenador paralelo, pero también es posible que se ejecuten en un solo procesador mediante de alguna técnica de simulación, por ejemplo, los hilos de Java se ejecutan cuasi–simultanemente en una sola máquina virtual de Java dando a cada hilo cierto tiempo de ejecución según algún algoritmo de planificación adecuado (dicha máquina virtual a su vez puede aprovechar de varios procesadores disponibles en el sistema).

La concurrencia describe un paralelismo potencial para la ejecución del programa.

## 5.1. Aplicación

¿Cuándo se usan programas concurrentes?

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

### 5.1.1. Indicadores

¿Cuáles son indicadores que sugieren un programa concurrente?

- el problema consiste de forma natural en gestionar eventos (asincronidad, “asynchronous programming”)
- 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 quieren terminar o suspender tareas una vez empezadas (controlabilidad, “controllability”)
- el problema tiene que cumplir restricciones temporales
- el problema requiere que varias tareas se ejecutan (cuasi) simultaneamente (programación reactiva, “reactive programming”)
- se quiere ejecutar un programa más rápido y los recursos están disponibles (explotación del paralelismo, “Exploitation of parallelism”)
- la solución del problema requiere más recursos que un sólo ordenador puede ofrecer (explotación de hardware distribuido)
- 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.

### 5.1.2. Recursos

Entro otros, posibles recursos son

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

El las prácticas solamente nos dedicaremos a los dos últimos puntos, sobre todo tratamos estructuras de datos como recursos que varios hilos quieres usar a la vez.

### 5.1.3. Ejemplos clásicos

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

- sistemas operativos
    - soportar operaciones casi–paralelas

  - proveer servicios a varios usuarios a la vez (sistemas multi-usuario, sin largos tiempos de espera)
  - gestionar recursos compartidos (por ejemplo, sistemas de ficheros)
  - reaccionar a eventos no predeterminados
- sistemas en 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
- sistema de reservas y compra (“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 el mismo recurso 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 (por ejemplo, la internet)
  - 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
  - queremos “aprovechar” del otro lado para acceder/intercambiar información (por ejemplo, documentos multimedia) y acción (por ejemplo, juegos sobre la red, juegos distribuidos)
  - se quiere incorporar los dispositivos distribuidos para realizar cálculos complejos (SETI) o controles remotos (casa inteligente)
- sistemas tolerantes a fallos
  - se vigila de forma concurrente el funcionamiento correcto de otra aplicación
- servicios distribuidos
  - varios clientes pueden conectarse a un servidor que les gestiona cierta petición
  - el sistema puede ser más complejo, por ejemplo, incluyendo delegación de servicios

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

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

Hoy día existen muchos APIs de tipo “middleware” que facilitan el desarrollo de aplicaciones distribuidas, por ejemplo, JavaRMI, CORBA, JINI etc. Dichos entornos de desarrollo mantienen muchos detalles al márgen del programador (y del usuario), es decir, se usa las capas bajas de forma transparente (por ejemplo, protocolos fiables de comunicación, iniciación de procesos remotos etc.).

### 5.1.4. Ejemplos no tan clásicos

- implementación de herramientas para el trabajo cooperativo en entornos distribuidos (por ejemplo: editor concurrente SubEthaEdit, herramientas para la programación extrema)
- aplicaciones en redes peer–to–peer sin servidores
- aplicaciones en redes adhoc, donde se forman redes de ordenadores de forma espontanea por su acercamiento geográfica
- juegos distribuidos sin cuello de botella de un servidor
- herramientas de teleformación con la posibilidad del trabajo en grupos a distancia

## 5.2. Implementación

Nos enfocamos solamente a programas escritos en lenguajes imperativos con concurrencia, comunicación, y sincronización explícita.

Como cualquier otra tarea de programación nos enfrentamos a los problemas de

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

## 5.3. Crítica

Entre las ventajas de la programación concurrente/distribuida en relación con el rendimiento se espera:

- que el programa se ejecute más rápido,
- si se usa los recursos de mejor manera, por ejemplo no deje recursos disponibles sin uso durante mucho tiempo (por ejemplo procesadores a disposición)
- y que el programa refleje o bien el modelo del problema real o bien la propia realidad

Sin embargo, también existen desventajas:

- se pierde tiempo en sincronizar procesos y comunicar datos entre ellos
- en el caso de multiplexación 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 no se produjera ningún progreso en el programa
- los sistemas pueden ser mucho más heterógenos
- la fiabilidad y disponibilidad de los recursos es muy diferente a un sistema secuencial
- 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 más complejo que el desarrollo de programas secuenciales
- la depuración de programas concurrentes es *muy difícil*, (por eso vale la pena de mantener una estricta disciplina en el desarrollo de programas concurrentes y basar la implementación en patrones de diseño bien estudiados)

# 6. Java

Este repaso a Java no es ni completo ni exhaustivo ni suficiente para programar en Java, debe servir solamente para refrescar conocimiento ya adquerido y para animar de profundizar el estudio del lenguaje con otras fuentes, por ejemplo, con la bibliografía añadida y los manuales correspondientes.

## 6.1. Hola mundo

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

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

## 6.2. 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++ o C# (por ejemplo, en su síntaxis y gran parte de sus metodologías), aunque también existen grandes diferencias (por ejemplo, en su no–uso o uso de punteros y la gestión de memoria).

```
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 pasan como un arreglo de cadenas de letras (`String`).

Java exige una disciplina estricta con sus tipos, es decir, el compilador controla siempre que puede 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.

## 6.3. Modificadores de clases

Se puede declarar clases con uno o varios de los siguientes modificadores para especificar ciertas propiedades (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 hayan implementado en una clase derivada los métodos que faltan
- `final` no se puede extender la clase
- `strictfp` obliga a la máquina virtual a 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 abstracta. Tampoco está permitida una clase abstracta con `strictfp`.

## 6.4. Comentarios

Existen tres posibilidades de escribir comentarios:

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

## 6.5. Tipos simples

| | |
|---|---|
| `boolean` | o bien `true` o bien `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 (por ejemplo, `NEGATIVE_INFINITY` etc.).

## 6.6. 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`), aunque las constantes no se pueden modificar en el transcurso del programa, pueden ser calculadas durante sus construcciónes; las 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 a 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 a la máquina virtual de Java que no use ningún tipo de cache para el miembro, así es más probable (aunque no garantizado) que varios hilos vean el mismo valor de una variable; declarando variables del tipo `long` o `double` como `volatile` aseguramos que las operaciones básicas sean atómicas (este tema veremos más adelante más en detalle)

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 sobreescribir 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 a 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 deriva en programas más eficientes.

## 6.7. Estructuras de control

Las estructuras de control son casi 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 que 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;
    }
  }
```

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

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

## 6.8. 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 (C++ tiene un concepto parecido con `typeid`)
- 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 (por ejemplo, los `[]`). Además Java no proporciona la posibilidad de sobrecargar operadores.

## 6.9. 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. Aunque `goto` y `const` aparecen en la lista arriba, no se usan en el lenguaje.

## 6.10. 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 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, por ejemplo, `X`, construye un miembro estático de otra clase, por ejemplo, `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 con C++ donde se tiene que liberar 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 la reinterpretación del tipo (“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 ) { ... }
```

## 6.11. Parámetros

Se pueden 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 y cuando se distingan 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 por valor. 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 métodos y pasarles como parámetros variables de tipos simples.

## 6.12. 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).

## 6.13. Arreglos (*Arrays*)

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

El código

```
int[] vector = new int[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 con C++ donde no hay una comprobación). Si se detecta un acceso fuera de los límites se produce una excepción `IndexOutOfBoundsException`. Dependiendo de las capacidades del compilador eso puede resultar en una pérdida de rendimiento.

Los arreglos son objetos implícitos que siempre conocen sus propias longitudes (`values.length`) (diferencia con C++ donde un arreglo no es nada más que un puntero) y que se comportan como clases finales.

No se pueden declarar los elementos de un arreglo como constantes (como es posible en C++), es decir, el contenido de los componentes siempre se puede modificar en un programa en Java.

## 6.14. `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 bien a otro constructor con `this(...)` o bien 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

## 6.15. Extender clases

Se puede crear nuevas clases a partir de la extención de clases ya existentes (en caso que no sean finales). Las nuevas clases se suelen llamar sub–clases o clases extendidas.

Una sub–clase heredará todas las propriedades de la clase súperior, aunque 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 sobreescribir métodos de la super–clase. Si se ha sobreescrito una cierta función, automáticamente todas las funciones con el mismo nombre aún con otras signaturas ya no están accesibles de modo directo.

Si se quiere ejecutar dentro de un método sobreescrito 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 sobreescritos. Se pueden 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 sobreescrito 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.

Si 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.

## 6.16. Clases dentro de clases

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

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

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

## 6.17. Clases locales

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

## 6.18. La clase `Object`

Todos los objetos de Java son extensiones de la clase `Object`. Los métodos públicos 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, se avisa al administrador de la memoria que ya no se usa dicho objeto, y se puede ejecutar código especial antes de que se libere la memoria
- `public String toString()` devuelvo una cadena describiendo el objeto

Las clases derivadas deben sobreecribir los métodos adecuadamente, por ejemplo el método `equals`, si se requiere una comparación binaria.

## 6.19. Clonar objetos

Lo veremos cuando nos haga falta.

## 6.20. 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 de 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, aunque no se haya descrito explícitamente.

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

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

Todos los métodos de una interfaz son implícitamente públicos y abstractos, aunque no se haya descrito 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 implementen.

Una clase puede implementar varias interfaces al mismo tiempo (aunque 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
  }
```

Las 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.

# 7. Hilos de Java

Un hilo es una secuencia de instrucciones que está controlada por un planificador que se comporta como un flujo de control secuencial. 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 problemas y *sorpresas* de dicho funcionamiento veremos más adelante.

En Java 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 hereda 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 al planificador de 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()` (puede ser que el tercer hilo siendo el programa principal está intercalado de vez en cuando)
- dicho comportamiento depende del sistema concreto con el cual se está trabajando

Prácticas: codificar los ejemplos y realizar un pingPONG perfecto.

# 8. Repaso: programación secuencial

Asumimos que tengamos solamente las operaciones aritméticas *sumar* y *subtraer* disponibles en un procesador fictício 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 se llega 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 que han llegado a la instrucción `a:`)
- se llega a la instrucción `g:` en algún momento

Tenemos que saber que las instrucciones atómicas son correctas, es decir, sabemos exactamente su significado, incluyendo todos los efectos segundarios posibles.

Luego usamos el concepto de inducción para comprobar el bucle. Sean $p_i$, $q_i$, y $r_i$ los contenidos de los registros `p`, `q`, y `r`, respectivamente.

La invariante cuya correción hay que comprobar con el concepto de inducción es entonces:

$$r_i + p_i \cdot q_i = p \cdot q$$

# 9. Primer algoritmo concurrente

Reescribimos el algoritmo secuencial para que "funcione" con dos procesos:

```
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 indeterminista 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.

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

**Ejemplo:** (mostrado en pizarra)

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 ejecuten las instrucciones.

Para comprobar si un programa concurrente es incorrecto basta con encontrar una intercalación de instrucciones que nos lleva 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 la mera enumeración si un programa concurrente es correcto bajo todas las ejecuciones posibles.

En la argumentación hasta ahora fue muy importante que las instrucciones se ejecutaran 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 o al revés sin analizar el concepto de memoria compartida detenidamente.

Ejemplo de Java: accesos a variables con más de 4 byte no son atómicos.

# 10. Abstracción

Un programa concurrente se puede ver como un conjunto de

- procesos que consisten en secuencias de instrucciones atómicas
- 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 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 tienen por que ser conocidas por el programador.

## 10.1. Instrucciones atómicas

Se considera instrucciones atómicas aquellas que está garantizado su correcto cumplimento independientemento de otras instrucciones posiblemente ejecutadas simultaneamente en el programa. A veces también nos referimos a instrucciones atómicas cuando el procesador es capaz de volver al estado producido

justamente antes de haber empezado la ejecución de la instrucción en el caso de que se produzca una interrupción, es decir, el efecto de la instrucción es nulo.

Un algoritmo secuencial impone un orden total en el conjunto de las instrucciones que establece mientras un algoritmo concurrente solamente especifica un orden parcial.

Aunque eso no es totalmente cierto en el caso de microprocesadores modernos que son capaces de reordenar las instrucciones para conseguir un mejor rendimiento del procesador. Estas reordenaciones y ejecuciones especulativos suelen mantener la semántica de la secuencia de operaciones, sin embargo, incluso en orden del acceso a la memoria principal puede ser cambiada.

El orden establecido se puede visualizar con grafos: los nodos representan las instrucciones atómicas, las aristas indican si una instrucción debe seguir la otra.

Dicho orden se puede determinar con las condiciones de Bernstein. Sea $L(S_k)$ el conjunto de variables que se *leen* durante la ejecución de las instrucciones en $S_k$. Sea $E(S_k)$ el conjunto de variables que se *escriben* durante la ejecución de las instrucciones en $S_k$.

Entonces, se pueden ejecutar los dos conjuntos de instrucciones $S_i$ y $S_j$ concurrentemente, siempre cuando se cumpla: $L(S_i) \cap E(S_j) = E(S_i) \cap L(S_j) = E(S_i) \cap E(S_j) = \emptyset$.

Dado que en un programa concurrente varios procesos actuan simultaneamente sobre las instrucciones, no se puede establecer un orden total.

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.

## 10.2. Regiones críticas

Una región o sección 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 deban ser alterados por otros procesos, porque es posible que su valor o contenido en el momento de lectura no sean validos; puede ser que estén en un estado transitorio. Sin embargo, si los accesos concurrentes solamente leen pueden estar permitidos (más sobre el tema veremos más adelante).

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 por parte del programador en cuanto a acceder a las variables críticas solamente con métodos adecuados.

## 10.3. Funcionamiento correcto

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

Más formalmente:

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

Un ejemplo es el cálculo de la raíz cuatrada, si $x$ es un número flotante (por ejemplo en el estándar IEEE) queremos que un programa que calcula la raíz, lo hace correctamente para todos los números $x \geq 0$. Para que los procesadores puedan usar una función total (que hoy día ya es parte de las instrucciones básicas de muchos procesadores), hay que incluir los casos que $x$ es negativo; para eso el estándar usa la codificación de `nan` (*not-a-number*). Calcular la raíz de un número negativo (o de `nan`) resulta en `nan`.

Se distingue los dos casos sobre todo porque el problema si un programa, dado una entrada, se para, no es calculable. O en otras palabras, no podemos "completar" siempre la función por calcular.

Para un programa secuencial existe solamente un orden total de las instrucciones atómicas (en el sentido que un procesador secuencial siempre sigue el mismo orden de las instrucciones), mientras que para un programa concurrente puedan existir varios órdenes.

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.

Entonces:

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.
- Los programas no deben estar basados en la suposición de que habrá un intercalado específico entre los hilos de parte del planificador.

# 11. Propriedades de programas concurrentes

## 11.1. Seguridad y vivacidad/viveza

Un programa concurrente puede fallar por varias razones, las cuales se pueden clasificar entre dos grupos de propiedades:

**seguridad:** Esa propriedad inidca que no está pasando nada malo en el programa, es decir, el programa no ejecuta instrucciones que no deba hacer ("safety property").

**vivacidad:** Esa propriedad indica que está pasando continuamente algo bueno durante la ejecución, es decir, el programa consigue algún progreso en sus tareas o en algún momento en el futuro se cumple una cierta condición ("liveness property").

Las propiedades de seguridad suelen ser algunas de las invariantes del programa que se tienen que introducir en las comprobaciones del funcionamiento correcto (por ejemplo, mediante inducción).

Ejemplos de propiedades de seguridad:

**Corrección:**

el algoritmo usado es correcto

**Exclusión mutua:** el acceso con exclusión mutua a regiones críticas está garantizado

**Sincronización:** los procesos cumplen con las condiciones de sincronización impuestos por el algoritmo

**Interbloqueo:** no se produce una situación en la cual todos los procesos participantes quedan atrapados en una espera a una condición que nunco se cumpla.

Ejemplos de propiedades de vivacidad:

**Inanición:** un proceso puede "morirse" por inanición ("starvation"), es decir, un proceso o varios procesos siguen con su trabajo pero otros nunca llegan a utilizar los recursos por ser excluidos de la competición por los recursos (por ejemplo en Java el uso de `suspend()` y `resume()` no está recomendado)

**Bloqueo activo:** puede ocurrir el caso que varios procesos están continuamente competiendo por un recurso de forma activa, pero ningúno de ellos lo consigue ("livelock")

**Cancelación:** un proceso puede ser terminado desde fuera sin motivo correcto, dicho hecho puede resultar en un bloqueo porque no se ha considerado la necesidad que el proceso debe realizar tareas necesarias para liberar recursos (por ejemplo, en Java el uso del `stop()` no está recomendado)

**Espera activa:** un proceso está comprobando continuamente una condición malgastando de esta manera tiempo de ejecución del procesador

## 11.2. Justicia entre procesos

Cuando los procesos compiten por el acceso a recursos compartidos se puede definir varios conceptos de justicia, por ejemplo:

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

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

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

**espera ordenada en tiempo:** si un proceso pide acceso, le será dado antes de todos los procesos que lo hayan 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 clara (lo veremos más adelante).

Normalmente se quiere que todos los procesos manifesten algún progreso en su trabajo. Sin embargo, eso no es una condición necesario en programas concurrentes; se puede vivir bien—en ciertas cituaciones—con algunos procesos "muertos", mientras no den otros problemas para el controlador (por ejemplo, llenar las tablas limitadas del sistema operativo). Siempre existe la posibilidad que el trabajo asignado a un proceso sea hecho por otro proceso dejando el primero en espera.

### 11.3. Espera activa de procesos

El algoritmo de Dekker (que vemos pronto) y sus derivados provocan una espera activa de los procesos cuando quieren acceder a 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 permanezcan en estos bucles de espera activa porque se está gastando potencia del procesador inútilmente.

Un método mejor consiste en 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.

### 11.4. Espera infinita o inanición de procesos

En programas concurrentes es posible que un proceso nunca llegue a hacer nada si el planificador o el control de los recursos compartidos respectivamente no permite que el proceso pueda cumplir con sus objetivos. Es decir, el proceso está sometido a una espera infinita (inanición).

Existen varias técnicas para evitar una 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 cuanto más tiempo un proceso tiene que esperar más alto se pone su prioridad con el fin que en algún momento su prioridad sea la más alta.
- Otra(s) técnica(s) se pide desarrollar en las tareas de programación.

## 12. Exclusión mutua a nivel bajo

Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control que permite la entrada de un proceso si el recurso está disponible y que prohibe la entrada de un proceso si el recurso está ocupado.

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

Entonces el código refleja una estructura como sigue:

```
...\\
entrance protocol
critical section
exit protocol
...\\
```

El protocolo de entrada y salida debe cumplir con las siguientes condiciones:

- solo un proceso debe obtener acceso a la sección crítica (acceso con exclusión mutua)
- un proceso debe obtener acceso a la sección crítica después de un tiempo de espera *finita*

Dependiendo de las capacidades del hardware la implementación de los protocolos de entrada y salida es más facil o más difícil, además las soluciones puedes ser más o menos eficiente.

Primero veremos que se puede realizar un protocolo seguro solamente con las instrucciones `load` y `store` de un procesador. Las soluciones no serán muy eficientes, especialmente si muchos procesos compiten por la sección crítica. Sin embargo, su desarrollo y la presentación de la solución ayuda en entender el problem principal.

Todos los microprocesadores modernos proporcionan instrucciones atómicas que permiten realizar los protocolos de forma más eficiente como veremos en las últimas secciones de este apartado.

## 12.1. Algoritmo de Dekker

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

### 12.1.1. Primer intento

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

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

- Está garantizada la exlusión mutua porque un proceso llega a su línea `c:` solamente si el valor de `v` corresponde a su identificación (que asumimos siendo única).
- Obviamente, los procesos pueden acceder al recurso solamente alternativamente, que puede ser inconveniente porque acopla los procesos fuertemente.
- Un proceso no puede entrar más de una vez seguido en la sección crítica.
- Si un proceso termina el programa (o no llega más por alguna razón a su línea `d:`, el otro proceso puede resultar bloqueado.
- La solución se puede ampliar fácilmente a más de dos procesos.

### 12.1.2. Segundo intento

Intentamos evitar la alternancia. Usamos para cada proceso una variable, `v0` para `P0` y `v1` para `P1` respectivamente, que indica si el correspondiente proceso está usando el recurso.

```
  P0                                P1
a:  loop                              loop
b:    wait until v1 equals false        wait until v0 equals false
c:    set v0 to true                    set v1 to true
d:    critical section                  critical section
e:    set v0 to false                   set v1 to false
f:    non-critical section              non-critical section
g:  endloop                           endloop
```

- Ya no existe la situación de la alternancia.
- 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.

¿Cuál es la intercalación maligna?

### 12.1.3. Tercer intento

Cambiamos el lugar donde se modifica la variable de control:

```
  P0                                P1
a:  loop                              loop
b:    set v0 to true                    set v1 to true
c:    wait until v1 equals false        wait until v0 equals false
d:    critical section                  critical section
e:    set v0 to false                   set v1 to false
f:    non-critical section              non-critical section
g:  endloop                           endloop
```

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

¿Cuál es la intercalación maligna?

### 12.1.4. Cuarto intento

Modificamos la instrucción `c:` para dar la oportunidad que el otro proceso encuentre su variable a favor.

```
  P0                                P1
a:  loop                              loop
b:    set v0 to true                    set v1 to true
c:    repeat                            repeat
d:      set v0 to false                   set v1 to false
e:      set v0 to true                    set v1 to true
f:    until v1 equals false             until v0 equals false
g:    critical section                  critical section
h:    set v0 to false                   set v1 to false
i:    non-critical section              non-critical section
j:  endloop                           endloop
```

- Se puede producir una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil (*livelock*)

¿Cuál es la intercalación maligna?

#### 12.1.5. Quinto intento

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
a:  loop                              loop
b:    set v0 to true                    set v1 to true
c:    loop                              loop
d:      if v1 equals false exit           if v0 equals false exit
e:      if v equals P1                    if v equals P0
f:        set v0 to false                   set v1 to false
g:        wait until v equals P0            wait until v equals P1
h:        set v0 to true                    set v1 to true
i:      fi                                fi
j:    endloop                           endloop
k:    critical section                  critical section
l:    set v0 to false                   set v1 to false
m:    set v to P1                       set v to P0
n:    non-critical section              non-critical section
o:  endloop                           endloop
```

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

### 12.2. Algoritmo de Lamport

o algoritmo de la panadería:

- cada proceso tira un ticket (que están ordenados en orden ascendente)
- cada proceso espera hasta que su valor del ticket sea el mínimo entro todos los procesos esperando

- el proceso con el valor mínimo accede la sección crítica

Observaciones:

- se necesita un cerrojo para acceder a los tickets
- el número de tickets no tiene límite
- los procesos tienen que comprobar continuadamente todos los tickets de todos los demás procesos

Por eso el algoritmo no es verdaderamente practicable (infinitos tickets y ineficiencia por el número elevado de comprobaciones).

## 12.3. Otros algoritmos

Existen algoritmos que resuelven el problema de la exclusión mutua. Una versión de Peterson del algoritmo de Dekker es más corto. Como vimos, el algoritmo de Lamport (algoritmo de la panadería) necesita muchas comparaciones de los tickets para $n$ procesos. Existe una versión de Peterson que usa solamente variables confinadas a cuatro valores. Otra posibilidad es al algoritmo de Eisenberg–McGuire. (mirad la bibliografía).

## 12.4. Ayuda con hardware

Si existen instrucciones más potentes (que los simples `load` y `store`) en el microprocesador se puede realizar la exclusión mutua más fácil.

Hoy casi todos los procesadores implementan un tipo de instrucción atómica que realiza algún cambio en la memoria al mismo tiempo que devuelve el contenido anterior de la memoria.

### 12.4.1. Comprobar–y–poner (*Test–and–set*)

La instrucción `test-and-set` (TAS) implementa una comprobación a cero del contenido de una variable en la memoria al mismo tiempo que varía su contenido en caso que la comprobación se realizó con el resultado verdadero.

Así se puede realizar la exclusión mutua con

```
Initially:  vi is equal false
            C  is equal true

a: loop
b:   non-critical section
c:   loop
d:     if C equals true            ; atomic test-and-set
e:       set C to false and exit
f:   endloop
g:   set vi to true
h:   critical section
i:   set vi to false
j:   set C to true
h: endloop
```

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

Teniendo solamente una variable para la sincronización de varios procesos el algoritmo arriba no garantiza una espera limitada de todos los procesos participando. ¿Por qué?

Para conseguir una espera limitada se implementa un protocolo de paso de tal manera que un proceso saliendo de su sección crítica da de forma explícita paso a un proceso esperando (en caso que tal proceso exista). (Prácticas)

### 12.4.2. Intercambiar (*exchange*)

La instrucción `exchange` (a veces llamado `read-modify-write`) intercambia un registro del procesador con el contenido de una dirección de la memoria en una instrucción atómica.

Así se puede realizar la exclusión mutua con

```
Initially:  vi is equal false
            C  is equal true

a: loop
b:   non-critical section
c:   loop
d:     exchange C and vi       ; atomic exchange
e:     if vi equals true exit
f:   endloop
g:   critical section
h:   exchange C and vi
i: endloop
```

Se observa lo mismo que en el caso anterior, no se garantiza una espera limitada. ¿Cómo se consigue? (Prácticas).

### 12.4.3. Decremento/incremento atómico (*fetch–and–increment*)

La instrucción `fetch-and-increment` aumenta el valor de una variable en la memoria y devuelve el resultado en una instrucción atómica.

Con dicha instrucción se puede realizar los protocolos de entrada y salida. ¿Cómo? (Prácticas).

### 12.4.4. Comparar–y–intercambiar (*compare–and–swap*)

La instrucción `compare-and-swap` (CAS) es una generalización de la instrucción `test-and-set`. La instrucción trabaja con dos variables, les llamamos `C` (de *compare*) y `S` (de *swap*) Se intercambia el valor en la memoria por `S` si el valor en la memoria es igual que `C`.

Es la operación que se usa por ejemplo en los procesadores de Intel y es la base para facilitar la concurrencia en la máquina virtual de Java 1.5 para dicha familia de procesadores.

Con dicha instrucción se puede realizar los protocolos de entrada y salida. ¿Cómo? (Prácticas).

Existe también una mejora del `CAS`, llamado *double-compare-and-swab*, que realiza dos CAS normales a la vez, el código, expresado a alto nivel, sería:

```
if  C1 equal to V1 and C2 equal to V2
  then
    set C1 to S1
    set C2 to S2
    return true
  else
    return false
```

# 13. Exclusión mutua a nivel alto

El concepto de usar estructuras de datos a nivel alto libera al programador de los detalles de su implementación. El programador puede asumir que las operaciones están implementadas 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 compitiendo.

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, por ejemplo, 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 haya ejecutado 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 sea 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 interrupción o porque se ha dado acceso al semáforo.

  **`signal()`** : Una vez se ha terminado el uso del recurso, el proceso lo señaliza al semáforo. Si queda algún proceso bloqueado en el semáforo uno de ellos sea 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 hubiera alguno bloqueado.

  Para despertar los procesos se puede implementar varias formas que se destinguen en sus grados de justicia, por ejemplo con un simple sistema tipo FIFO.

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 sea ejecutada por otro proceso.
- Con simples semáforos no se puede imponer un orden en los procesos accediendo a diferentes recursos.

Si existen en un entorno solamente semáforos binarios, se puede simular un semáforo general usando dos semáforos binarios y un contador, por ejemplo, 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 imponer 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 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 (así funciona Java con sus bloques sincronizados):

```
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 llame a otra parte del programa (por ejemplo, un procedimiento o función) que a su vez contenga una región crítica. Si esta región está controlada por la misma variable V el proceso obtiene automáticamente también acceso a dicha región.

Los 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 simples no se puede realizar tal control. Hace falta otra construcción, 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 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 (en Java se puede implementar por ejemplo con una construcción de bucle con `try`).

Las regiones críticas condicionales funcionan internamente de la siguiente manera:

1. un proceso que quiere entrar en 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 en 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 a 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 derivar en muchas comprobaciones de la condición, todos en modo exclusivo, y puede causar perdidas de eficiencia. En ciertas circunstancias 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 las 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 permitido 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 de 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 (dicha técnica se refleja en Java con `wait()` y `notify()`).

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 (por ejemplo Java usa el concepto de monitores para realizar el acceso mutuamente exclusivo a sus objetos).

El uso de monitores es bastante costoso, porque se pierde eficiencia por tanto bloqueo de los prosesos. Por eso se intenta aprovechar de primitivas más potentes para aliviar este problema, como vemos en la siguiente sección.

# 14. Bloqueo

Un bloqueo se produce cuando un proceso está esperando 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 $P_0$ accede con exito a $r_1$ y $P_1$ accede con exito a $r_0$, ambos se quedan atrapados intentando tener acceso al otro recurso.

Cuatro condiciones se tienen que cumplir para que sea posible que se produzca 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 solo permiten ser usados por menos procesos que lo intentan al mismo tiempo
4. existe una cadena circular entre peticiones de procesos y asignaciónes de recursos

Un problema adicional con los bloqueos es que es posible que el programa siga funcionando correctamente según la definición, es decir, el resultado obtenido es el resultado deseado, aún cuando algunos de sus procesos están bloqueados durante la ejecución (es decir, se produjo solamente un bloque parcial).

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

## 14.1. Detectar y actuar

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

Más preciso, se define el grafo de asignación 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 eliminan las aristas entre proceso y recurso y al revés cuando el proceso ya no usa el recurso.

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

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

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

Sin embargo, las técnicas pueden dar como resultado que el programa no avance, incluso, el programa se puede quedar atrapado haciendo trabajo inútil: crear situaciones de bloqueo y abortar procesos continuamente.

## 14.2. Evitar

El sistema no da permiso de acceso a recursos si es posible que el proceso se bloquee en el futuro. Un método es el algoritmo del banquero (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 liberan 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 acceso 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.

## 14.3. Prevenir

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

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 (por ejemplo, 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 pida 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 aborte 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 ordenan los recursos línealmente y se fuerza a los procesos que accedan a los recursos en el orden impuesto. Así es imposible que se produzca un ciclo.

En las prácticas aplicamos dicho método para prevenir bloqueos en la lista concurrente.

# 15. Problema del productor y consumidor

El problema del productor y consumidor es un ejemplo clásico de programa concurrente y consiste en la situación siquiente: de una parte se produce algún producto (datos en nuestro caso) que se coloca 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 de lo que los está produciendo el productor. Más concretamente:

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 de lo que el productor produce), 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, lo 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.

Queremos ampliar el problema introduciendo más productores y más consumidores que trabajen 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 siñala 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. JSR166

La página inicial del proyecto JSR166 (*Java Specification Requests*) se encuentra bajo `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`

Su proposito principal es facilitar herramientas para la programación concurrente con Java y está planeado para estar disponible en la Version 1.5 de Java.

Seguimos un poco con las transparencias que explican el funcionamiento de dicho paquete de utilidades `java.util.concurrent` (*Doug Lea*): `./jsr166.pdf`

# 17. Arquitecturas que soportan la concurrencia

Se suele distinguir concurrencia

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

Una clasificación clásica de ordenadores paralelos es:

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

Hoy día, concurrencia a grano fino es estándar en los microprocesadores. El la familia de los procesadores de Intel, por ejemplo, existen las instrucciones MMX, SSE, y SSE2 que realizen según la clasificación SIMD operaciones en varios registros en paralelo. Ya están en el marcado los procesadores con dos coros, es decir, se puede programar con 4 procesadores virtuales que su vez pueden ejecutar 4 hilos independientes.

La programación paralela y concurrente (y con pipeline) se revive actualmente en la programación de la GPU (graphics processing units) que son procesadores especializados para su uso en tarjetas gráficas que cada vez se usa más para otros fines de cálculo numérico.

## 17.1. Conmutación

**multi–programación o *multi–programming*:** los procesos se ejecutan en hardware distinto

**multi–procesamiento o *multi–processing*:** Se aprovecha de la posibilidad de multiplexar varios procesos en un solo procesador.

**multi–tarea o *multi–tasking*:** El sistema operativo (muchas veces con la ayuda de hardware específico) realiza la ejecución de varios procesos de forma cuasi–paralelo distribuyendo el tiempo disponible a las secuencias diferentes (*time–sharing system*) de diferentes usuarios (con los debidas medidas de seguridad).

La visión de ‘computación en la red’ no es nada más que un gran sistema MIMD.

Existen dos puntos de vista relacionados con el mecanismo de conmutación

- el mecanismo de conmutación es *independiente* del programa concurrente (eso suele ser el caso en sistemas operativos),
- el mecanismo de conmutación es *dependiente* del programa concurrente (eso suele ser el caso en sistemas en tiempo real),

En el segundo caso es imprecindible incluir dicho mecanismo en el análisis del programa.

## 17.2. Memoria compartida

Sin una memoria compartida no existe concurrencia.

Existen varios tipos de arquitecturas de ordenadores que son diseñadas especialmente para la ejecución de programas concurrentes o paralelos con una memoria compartida (por ejemplo los proyectos NYU, SB-PRAM, o Tera; *shared memory processing*), más se explicó en clase.

Muchas ideas de estos proyectos se encuentra hoy día en los microprocesadores modernos, sobre todo en los protocolos que controlan la coherencia de los cachés.

Sin embargo, no hace falta que se ejecute un programa en unidades similares para obtener concurrencia. La concurrencia está presente también en sistemas heterógenos, por ejemplo, aquellos que solapan el trabajo de entrada y salida con el resto de las tareas (discos duros).

La comunicación y sincronización entre procesos funciona

- mediante una memoria compartida (*shared memory*) a la cual pueden acceder todos los procesadores a la vez o
- mediante el intercambio de mensajes usando una red conectando los diferentes procesadores u ordenadores, es decir, procesamiento distribuido (*distributed processing*).

Sin embargo, siempre hace falta algún tipo de memoria compartida para realizar la comunicación entre procesos, solamente que en algunos casos dicha memoria no es accesible en forma directa por el programador.

También existen mezclas de todo tipo de estos conceptos, por ejemplo, un sistema que use multi–procesamiento con hilos y procesos en cada procesador de un sistema distribuido simulando una memoria compartida al nivel de la aplicación.

# 18. Comunicación y sincronización

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

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

## 18.1. Metódos de comunicación

Para cualquier tipo de comunicación 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 (*shared memory*),
2. mandar mensajes (*message passing*),
3. lanzar procedimientos remotos (*remote procedure call* RPC).

La comunicación no tiene que ser síncrona 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.

## 18.2. Canal de comunicación

Una comunicación entre procesos sobre algún canal físico puede ser no fiable en los sistemas.

Se puede usar el canal

- para mandar paquetes individuales del mensaje (por ejemplo protocolo UDP del IP)
- para realizar flujos de datos (por ejemplo protocolo TCP de IP)

Para los canales de paquetes, existen varias posibilidades de fallos:

1. se pierden mensajes

2. se cambia el orden de los mensajes
3. se modifican mensajes
4. se añaden mensajes que nunca fueron mandados

Existen técnicas para superar los problemas:

1. protocolo de recepción (¿Cuándo se sabe que haya llegado el último mensaje?)
2. enumeración de los mensajes
3. uso de código de correción de errores (CRC)
4. protocolo de autentificación

Los canales que realizan flujos de datos suelen tener implementado un "pila de protocolos" (*protocol stack*) para garantizar (hasta cierto punto) la correcta transmisión de los datos.

Existen protocolos de transmisión de paquetes que no necesitan un canal de retorno pero que garantizan la distribución de los mensajes bajo leves condiciones al canal (*digital fountain codes*), más se explicó en clase.

# 19. Programación orientada a objetos

Programar con objetos tiene muchas ventajas que no repetiremos 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 la 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 mensajes (de otros objetos)
- mandar mensajes (a otros objetos)
- crear/inicializar el objeto

Normalmente también se tiene que destruir un objeto al final de su tiempo de vida. Aunque eso no se tiene que hacer necesariamente de forma explícita.

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

- El modelo de objetos activos describe un objeto como una entidad con vida propia que actua cada vez que recibe 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.

Dicho concepto puede estar implementada en varios niveles, por ejemplo, aplicando *pools* de hilos trabajadoras que, entre si, simulan el comportamiento de los objetos activos en el sistema.

# 20. Patrones de diseño para aplicaciones concurrentes

Los patrones de diseño para el desarrollo de software representan una herramienta para facilitar la producción de aplicaciones más robustos y más reusables. Se intenta plasmar los conceptos que se encuentran frecuentemente en las aplicaciones en algún tipo de *código generico*.

Un concepto muy parecido a los patrones de diseño se encuentra en la matemática y en la teoría de los algoritmos, por ejemplo:

**técnicas de pruebas matemáticas:**

- comprobación directa
- inducción
- contradicción
- contra–ejemplo
- comprobación indirecta
- diagonalización
- reducción
- y más

**paradigmas de desarrollo de algoritmos:**

- búsqueda exhaustiva
- búsqueda binaria
- divide–y–vencerás
- ramificación-y–poda
- barrido
- iteración
- perturbación
- amortización
- algoritmos evolutivos
- y más

En la continuación veremos unos patrones de diseño útiles para la programación concurrente.

## 20.1. Reactor

El patrón *reactor* es conocido también como *dispatcher* o *notifier*.

Se usa cuando una aplicación

- que gestióna eventos
- debe reaccionar a varias peticiones cuasi simultaneamente,
- pero las procesa de modo síncrono y en el orden de llegada.

Ejemplos:

- servidores con multiples clientes
- interfaces al usuario con varias fuentes de eventos
- servicios de transacciones
- *centralita*

Comportamiento exigido:

- la aplicación no debe bloquear innecesariamente otras peticiones mientras se está gestionando una petición
- debe ser fácil incorporar nuevos tipos de eventos y peticiones
- la sincronización debe ser escondida para facilitar la implementación de la aplicación

Posible solución:

- se espera en un bucle central a todos los eventos que pueden llegar
- una vez recibido un evento se traslada su procesamiento a un gestor específico de dicho tipo de evento
- el reactor permite añadir/quitar gestores para eventos

Detalles de la implementación:

- Bajo Unix y (parcialmente) bajo Windows se puede aprovechar de la función `select()` para el bucle central.
- Si los gestores de eventos son procesos independientes hay que evitar posibles interbloqueos o estados erroneos si varios gestores trabajan con un estado común.
- Se puede aprovechar del propio mecanismo de gestionar eventos para lanzar eventos que provoquen que el propio *reactor* cambie su estado.
- Java no dispone de un mecanismo apropriado para emular el `select()` de Unix.

## 20.2. Proactor

Se usa cuando una aplicación

- que gestióna eventos
- debe actuar en respuesta a varias peticiones casi simultaneamente y
- debe procesar los eventos de modo asíncrono notificando la terminación adecuadamente

Ejemplos:

- servidores para la Red
- interfaces al usuario para tratar componentes con largos tiempos de cálculo
- *contestador automático*

Comportamiento exigido (igual como en el caso del reactor):

- la aplicación no debe bloquear innecesariamente otras peticiones mientras se está gestionando una petición
- debe ser fácil incorporar nuevos tipos de eventos y peticiones
- la sincronización debe ser escondida para facilitar la implementación de la aplicación

Posible solución:

- se divide la aplicación en dos partes: operaciones de larga duración (que se ejecutan asíncronamente) y administradores de eventos de terminación para dichas operaciones
- con un iniciador se lanza cuando haga falta la operación compleja
- las notificaciones de terminación se almacena en una cola de eventos que a su vez el administrador está vaciando para notificar la aplicación de la terminación del trabajo iniciado
- el proactor permite añadir/quitar gestores para operaciones y administradores

Detalles de la implementación:

- muchas veces basta con un solo proactor en una aplicación que se puede implementar a su vez como *singleton*.
- se usa varios proactores en caso de diferentes prioridades (de sus colas de eventos de terminación)
- se puede realizar un bucle de iniciación/terminación hasta que algún tipo de terminación se haya producido (por ejemplo transpaso de ficheros en bloques y cada bloque de modo asíncrono.
- la operación asíncrona puede ser una operación del propio sistema operativo

## 20.3. Ficha de terminación asíncrona

El patrón *asynchronous completion token* es conocido también como *active demultiplexing* o *magic cookie*.

Se usa cuando una aplicación

- que gestióna eventos
- debe actuar en respuesta a sus propias peticiones
- de modo asíncrono después de ser notificado de la terminación del procesamiento de la petición

Ejemplos:

- interacción compleja en un escenario de commercio electrónico (relleno de formularios, suscripción a servicios)
- interfaces al usuario con diálogos no bloqueantes
- *contestador automático*

Comportamiento exigido:

- se quiere separar el procesamiento de respuestas a un servicio
- se quiere facilitar un servicio a muchos clientes sin mantener el estado del cliente en el servidor

Posible solución:

- la aplicación manda con su petición un ficha indicando como hay que procesar después de haber recibido un evento de terminación de la petición
- la notificación de terminación incluye la ficha original
-

Detalles de la implementación:

- las fichas suelen incorporar una identificación
- las fichas pueden contener directamente punteros a datos o funciones
- en un entorno más heterógeno se puede aprovechar de objetos distribuidos (CORBA)
- hay que tomar medidas de seguridad para evitar el proceso de fichas no–deseados
- hay que tomar medidas para el caso de perder eventos de terminación

## 20.4. Aceptor–Conector

Se usa cuando una aplicación

- necesita establecer una conexión entre una pareja de servicios (por ejemplo, ordenadores en una red)
- donde el servicio sea transparente a las capas más altas de la aplicación
- y el conocimiento de los detalles de la conexión (activo, pasivo, protocolo) no son necesarios para la aplicación

Ejemplos:

- los super–servicios de unix (`inetd`)
- usando `http` para realizar operaciones (CLI)

Comportamiento exigido:

- se quiere esconder los detalles de la conexión entre dos puntos de comunicación
- se quiere un mecanismo flexible en la capa baja para responder eficientemente a las necesidades de aplicaciones para que se puedan jugar papeles como servidor, cliente o ambos en modo pasivo o activo
- se quiere la posibilidad de cambiar, modificar, o añadir servicios o sus implementaciones sin que dichas modificaciones afecten directamente a la aplicación

Posible solución:

- se separa el establecimiento de la conexión y su inicialización de la funcionalidad de la pareja de servicios (*peer services*), es decir, se usa una capa de transporte y una capa de servicios
- se divide cada pareja que constitúe una conexión en una parte llamada aceptor y otra parte llamada conector
- la parte aceptora se comporta pasiva esperando a la parte conectora que inicia activamente la conexión
- una vez establecida la conexión los servicios de la aplicación usan la capa de transporte de modo transparente

Detalles de la implementación:

- muchas veces se implementa un servicio par–en–par (*peer–to–peer*) donde la capa de transporte ofrece una pareja de conexiones que se puede utilizar independientemente en la capa de servicios, normalmente una línea para escribir y otra para recibir
- la inicialización de la capa de transporte se puede llevar a cabo de modo síncrono o asíncrono, es decir, la capa de servicios queda bloqueada hasta que se haya establecido la conexión o se usa un mecanismo de notificación para avisar a la capa de servicios del establecimiento de la conexión
- es recomendado de usar el modo síncrono solamente cuando: el retardo esperado para establecer la conexión es corto o la aplicación no puede avanzar mientras no tenga la conexión disponible
- muchas veces el sistema operativo da soporte para implementar este patrón, por ejemplo, conexiones mediante sockets
- se puede aprovechar de la misma capa de transporte para dar servicio a varias aplicaciones a la vez

## 20.5. Guardián

El patrón *guard* es conocido también como *scoped locking*, *synchronized block* o *execute–around–object*.

Se usa cuando una aplicación

- contiene procesos (hilos) que se ejecutan concurrentemente y
- hay que proteger bloques de código con un punto de entrada pero varios puntos de salida
- para que no entren varios hilos a la vez.

Ejemplos:

- cualquier tipo de protección de secciones críticas

Comportamiento exigido:

- se quiere que un proceso queda bloqueado si otro proceso ya ha entrado en la sección crítica, es decir, ha obtenido la llave exclusiva de dicha sección
- se quiere que independientemente del método usado para salir de la sección crítica (por ejemplo uso de `return`, `break` etc.) se devuelve la llave exclusiva para la región

Posible solución:

- se inicializa la sección crítica con un objeto de guardia que intenta obtener una llave exclusiva
- se aprovecha de la llamada automática de destructores para librar la sección crítica, es decir, devolver la llave

Detalles de la implementación:

- Java proporciona el guardián directamente en el lenguaje: métodos y bloques sincronizados (`synchronized`)
- hay que prevenir auto–bloqueo en caso de llamadas recursivas
- hay que tener cuidado con interrupciones forzadas que circundan el flujo de control normal
- porque el guardián no está usado en la sección crítica, el compilador puede efectuar ciertos mensajes de alerta y — en el caso peor — un optimizador puede llegar a tal extremo eliminando el objeto
- para facilitar la implementación de un guardián en diferentes entornos (incluyendo situaciones secuenciales donde el guardián efectivamente no hace nada) se puede aprovechar de estrategias de polimorfismo o de codificación con plantillas para flexibilizar el guardián (el patrón así cambiado se llama a veces: *strategized locking*)

## 20.6. Interfaz segura para multi–hilos

*thread–safe interface*

Se usa cuando una aplicación

- usa muchos hilos que trabajan con los mismos objetos

- y se quiere minimizar el trabajo adicional para obtener y devolver la llave que permite acceso en modo exclusivo

Ejemplos:

-

Comportamiento exigido:

- se quiere evitar auto–bloqueo debido a llamadas del mismo hilo para obtener la misma llave
- se quiere minimizar el trabajo adicional

Posible solución:

- se aprovecha de las interfaces existentes en el lenguaje de programación para acceder a los componentes de una clase
- cada hilo accede solamente a métodos públicos mientras todavía no haya obtenido la llave
- dichos métodos públicos intentan obtener la llave cuanto antes y delegan después el trabajo a métodos privados (protegidos)
- los métodos privados (o protegidos) asumen que se haya obtenido la llave

Detalles de la implementación:

- los monitores de Java proporcionan directamente un mecanismo parecido al usuario, sin embargo, ciertas clases de Java (por ejemplo, tablas de dislocación (*hash tables*)) usan internamente este patrón por razones de eficiencia
- hay que tener cuidado de no corromper la interfaz, por ejemplo, con el uso de métodos amigos (`friend`) que tienen acceso directo a partes privadas de la clase
- el patrón no evita bloqueo, solamente facilita una implementación más transparente

## 20.7. Aviso de hecho

El patrón *double–checked locking optimization* es conocido también como *lock hint*.

Se usa cuando una aplicación

- usa objetos (clases) que necesitan una inicialización única y exclusiva (patrón *Singleton*)
- que no se quiere realizar siempre
- sino solamente en caso de necesidad explícita y
- que puede ser realizada por cualquier hilo que va a usar el objeto por primera vez.

Ejemplos:

-

Comportamiento exigido:

- se quiere un trabajo mínimo en el caso que la inicialización ya se ha llevado a cabo
- se quiere que cualquier hilo puede realizar la inicialización
-

Posible solución:

- se usa un guardián para obtener exclusión mutua
- se comprueba dos veces si la inicialización ya se ha llevado a cabo: una vez antes de obtener la llave y una vez después de haber obtenido la llave

Detalles de la implementación:

- hay que marcar la bandera que marca si la inicialización está realizada como volátil (`volatile`) para evitar posibles optimizaciones del compilador
- el acceso a la bandera tiene que ser atómico

## 20.8. Objetos activos

El patrón *active object* es conocido también como *concurrent object*.

Se usa cuando una aplicación

- usa varios hilos y objetos
- donde cada hilo puede realizar llamadas a métodos de varios objetos que a su vez se ejecutan en hilos separados

Ejemplos:

- comportamiento de camarero y cocina en un restaurante

Comportamiento exigido:

- se quiere una alta disponibilidad de los métodos de un objeto (sobre todo cuando no se espera resultados inmediatos, por ejemplo, mandar mensajes)
- se quiere que la sincronización necesaria para involucrar los métodos de un objeto sea lo más transparente que sea posible
- se quiere una explotación transparente del paralelismo disponible sin programar explicitamente planificadores en la aplicación

Posible solución:

- para cada objeto se separa la llamada a un método de la ejecución del código (es decir, se usa el patrón *proxy*)

- la llamada a un método (que se ejecuta en el hilo del cliente) solamente añade un mensaje a la lista de acciones pendientes del objeto
- el objeto ejecuta con la ayuda de un planificador correspondiente las acciones en la lista
- la ejecución de las tareas no sigue necesariamente el orden de pedidos sino depende de las decisiones del planificador
- la sincronización entre el cliente y el objeto se realiza básicamente sobre el acceso a la lista de acciones pendientes

Detalles de la implementación:

- para devolver resultados existen varias estrategias: bloqueo de la llamada en el proxy, notificación con un mensaje (interrupción), uso del patrón futuro (se deposita el objeto de retorno a la disposición del cliente)
- debido al trabajo adicional el patrón es más conveniente para objetos gruesos, es decir, dondo el tiempo de cálculo de sus métodos por la frecuencia de sus llamadas es largo
- se tiene que tomar la decisión apropriada: uso de objetos activos o uso de monitores
- se puede incorporar temporizadores para abordar (o tomar otro tipo de decisiones) cuando una tarea no se realiza en un tiempo máximo establecido

## 20.9. Monitores

El patrón *monitor object* es conocido también como *thread–safe passive object*.

Se usa cuando una aplicación

- consiste en varios hilos
- que actuan sobre el mismo objeto de modo concurrente

Ejemplos:

- colas de pedido y colas de espera en un restaurante tipo comida rápida

Comportamiento exigido:

- se protege los objetos así que cada hilo accediendo el objeto vea el estado apropiado del objeto para realizar su acción
- se quiere evitar llamadas explícitas a semáforos
- se quiere facilitar la posibilidad que un hilo bloqueado deje el acceso exclusivo al objeto para que otros hilos puedan tomar el mando (aún el hilo queda a la espera de re–tomar el objeto de nuevo)
- si un hilo suelta temporalmente el objeto, este debe estar en un estado adecuado para su uso en otros hilos

Posible solución:

- se permite el acceso al objeto solamente con métodos sincronizados

- dichos métodos sincronizados aprovechan de una sola llave (llave del monitor) para encadenar todos los accesos
- un hilo que ya ha obtenido la llave del monitor puede acceder libremente los demás métodos
- un hilo reestablece en caso que puede soltar el objeto un estado de la invariante del objeto y se adjunta en una cola de espera para obtener acceso de nuevo
- el monitor mantiene un conjunto de condiciones que deciden los casos en los cuales se puede soltar el objeto (o reanuar el trabajo para un hilo esperando)

Detalles de la implementación:

- los objetos de Java implícitamente usan un monitor para administrar las llamadas a métodos sincronizados
- hay que tener mucho cuidado durante la implementación de los estados invariantes que permiten soltar el monitor temporalmente a la reanuación del trabajo cuando se ha cumplido la condición necesaria
- hay que prevenir el posible bloqueo que se da por llamadas intercaladas a monitores de diferentes objetos: se suelta solamente el objeto de más alto nivel y el hilo se queda esperando en la cola de espera (un fallo común en Java)

## 20.10. Mitad–síncrono, mitad–asíncrono

Se usa cuando una aplicación

- tiene que procesar servicios síncronos y asíncronos a la vez
- que se comunican entre ellos

Ejemplos:

- administración de dispositivos controlados por interrupciones
- unir capas de implementación de aplicaciones que a nivel bajo trabajan en forma asíncrono pero que hacia ofrecen llamadas síncronas a nivel alto (por ejemplo, `read/write` operaciones a trajetas de red)
- organización de mesas en restaurantes con un camarero de recepción

Comportamiento exigido:

- se quiere ofrecer una interfaz síncrona a aplicaciones que lo desean
- se quiere mantener la capa asíncrona para aplicaciones con altas prestaciones (por ejemplo, ejecución en tiempo real)

Posible solución:

- se separa el servicio en dos capas que están unidos por un mecanismo de colas
- los servicios asíncronos pueden acceder las colas cuando lo necesitan con la posibilidad que se bloquea un servicio síncrono mientras tanto

Detalles de la implementación:

- hay que evitar desbordamiento de las colas, por ejemplo, descartar contenido en ciertas ocaciones, es decir, hay que implementar un control de flujo adecuado para la aplicación
- se puede aprovechar de los patrones *objetos activos* o *monitores* para realizar las colas
- para evitar copias de datos innesesarias se puede usar memoria compartida para los datos de las colas, solamente el control de flujo está separado

## 20.11. Líder–y–Seguidores

*leader–followers*

Se usa cuando una aplicación

- tiene que reaccionar a varios eventos a la vez y
- no es posible o conveniente inicializar cada vez un hilo para cada evento

Ejemplos:

- procesamiento de transacciones en tiempo real
- colas de taxis en aeropuertos

Comportamiento exigido:

- se quiere una distribución rápida de los eventos a hilos ya esperando
- se quiere garantizar acceso con exclusión mutua a los eventos

Posible solución:

- se usa un conjunto de hilos organizados en una cola
- el hilo al frente de la cola (llamado líder) procesa el siguiente evento
- pero transforma primero el siguiente hilo en la cola en nuevo líder
- cuando el hilo ha terminado su trabajo se añade de nuevo a la cola

Detalles de la implementación:

- se los eventos llegan más rápido que se pueden consumir con la cola de hilos, hay que tomar medidas apropriadas (por ejemplo, manejar los eventos en una cola, descartar eventos etc.)
- para aumentar la eficiencia de la implementación se puede implementar la cola de hilos esperando como un pila
- el acceso a la cola de seguidores tiene que ser eficiente y robusto

# 21. Concurrencia en memoria distribuida

## 21.1. 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 distribuidos 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 buen método de sincronizar procesos o trabajos, respectivamente. Existen muchas variantes de implementaciones de paso de mensajes. Destacamos unas características.

### 21.1.1. Tipos de sincronización

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

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

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

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

### 21.1.2. Identificación del otro lado

Se pueden distinguir varias posibilidades en cómo dos procesos envían y reciben sus mensajes:

- se usan nombres únicos para identificar tanto el remitente como el receptor
  entonces ambas partes tienen que especificar exactamente con que proceso quieren comunicarse
- solo el remitente especifica el destino, al receptor no le importa quién ha enviado el mensaje (cierto tipo de sistemas cliente/servidor)
- a ninguna de las dos partes le interesa cual será el proceso al otro lado, el remitente envía su mensaje a un buzón de mensajes y el receptor inspecciona su buzón de mensajes

### 21.1.3. Prioridades

Para el paso de mensajes su usa muchas veces el concepto de un canal entre el remitente y el receptor o también entre los buzones de mensajes y sus lectores. Dichos canales no tienen que existir realmente en la capa física de la red de comunicación.

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

# 22. Terminación de programas

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

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

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

Un programa concurrente termina cuando todas sus partes secuenciales han terminado.

Si el sistema distribuido contiene un procesador central que siempre está monitorizando a 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 en especial los canales de comunicación se resisten a la inspección y pueden contener mensajes aún no recibidos.

## 22.1. Detección de terminación

Asumimos el siguiente modelo de sistema distribuido:

- El sistema es fiable, es decir, ni los procesos/procesadores ni 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 son iniciados por un proceso ya iniciado.

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

- se envían mensajes para realizar las tareas del programa
- si un proceso recibe un mensaje lo tiene que procesar
- se envían señales para realizar la detección de terminación
- cuando se decide la terminación de un proceso (por la razón que sea) no envía más mensajes a los demás, sin embargo, si recibe de nuevo un mensaje reanuda el trabajo
- los canales para los mensajes y las señales existen siempre en pares y los canales de las 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 rara considerando los muchos problemas que se pueden solucionar con estrategias tipo divide-y-vencerás.

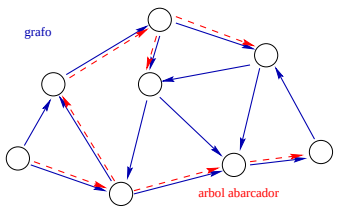
La detección de la terminación resulta fácil: una hija en el arbol manda y su madre que ha terminado cuando haya recibido el mismo mensaje de todas sus hijas y cuando se ha decido terminar también.

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 de 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 se desea terminar un proceso, se envían 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 y 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.




El algoritmo de determinación de terminación procede entonces como sigue:

- cuando un proceso decide terminar, envía señales según los valores déficit de todas sus aristas 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 que en el caso del árbol sencillo.

# 23. Tareas de programación

Las tareas de programación parecen simples leyendo su descrición, pero no son tan simples pensando en su realización.

## 23.1. Empezando

1. Consigue el “Hola Mundo” en Java.
2. Consigue un “Hola Mundo, soy hilo ...” usando varios hilos (con la clase `Thread` y también con la interfaz `Runnable`).
3. Mide cuantos hilos se puede lanzar simultaneamente en diferentes sistemas.
4. Mide el tiempo que un sólo hilo necesita para escribir por ejemplo 100000 veces "Hola Mundo", y cuanto tiempo necesitan por ejemplo 1000 hilos distribuyendo el trabajo entre ellos.
5. cambia el “trabajo que realiza un hilo” (escribir a consola) por algo que no tenga salida, observa las deferencias comparándolo con los resultados de antes.

Se observerá: se puede lanzar unos miles de hilos a la vez, el numero exacto depende de los recursos del ordenador, del sistema operativo, de la máquina virtual, y del entorno de programación.

Se observerá: muchos hilos necesitan en un sólo procesador más

## 23.2. PingPONG

1. Implementa el `pingPONG` perfecto. Ten los siguientes detalles en cuenta:
   - Experimenta con los diferentes intentos presentados en los apuntes.
   - Desarrolla una solución con las siguientes propiedades:
     *a*) Usa tres hilos (un hilo para el programa principal, o el árbitro, y un hilo para cada jugador).
     *b*) El árbitro inicia el juego (mensaje previo a la pantalla).
     *c*) Los jugadores producen sus `ping`s y `PONG`s alternamente.
     *d*) El árbitro termina el juego después de cierto tiempo (mensaje previo a la pantalla).
     *e*) Los jugadores realizen como mucho un intercambio de pelota más.
     *f*) Ambos jugadores/hilos terminan (mensaje previo a la pantalla).
     *g*) El árbitro escribe el último mensaje.
     *h*) El programa termina.
   - Observa la diferencia entre el uso de `notify()` y `notifyAll()`, sobre todo respecto a despertas “inútiles” de hilos.
2. Implementa el `pingPONG` entre dos ordenadores.
   - Asume que los IPs están conocidos.
   - Duplica la salida en los tres ordenadores, cada uno pone un prefijo delante, por ejemplo, `arbitro:`, `jugar rojo:`, y `jugador azul`.
3. Amplía el programa para que funcione como un `pingPANGpongPUNG`, es decir, con 4 jugadores.

4. Amplía el programa para que genere tantos jugadores (hilos) como se desea y genera una tabla de tiempos de ejecución incluyendo también el tiempo de ejecución con el caso del mismo programa usando un solo hilo (que entonces imprima solamente `ping`).

5. ¿Cuál sería una implementaciión "perfecta" en la cual se despierta solamente al hilo que tiene que jugar en este instante?

## 23.3. Planificación con prioridades

Implemente una aplicación con tres tipos de procesos/hilos con diferentes prioridades (digamos $A$, $B$ y $C$) que quieren acceder a un recurso.

1. ¿Cómo implementarías el control del planificador para que todos los procesos tengan acceso al recurso con la siguiente forma de justicia: dentro de la misma prioridad el acceso se realiza en orden de pedido y entre los diferentes prioridades se distribuye los accesos para que a lo largo del tiempo por lo menos unos 60 % de los accesos son para los procesos de tipo $A$, 30 % para los del tipo $B$ y 10 % para los del tipo $C$? (Ayuda: un planificador sabe contar).

2. Razona si tu solución garantiza una espera *finita* para todos los procesos pidiendo acceso al recurso.

## 23.4. Exclusión mutua a nivel bajo

Estos ejercicios se puede realizar sin Java.

1. Implementa los protocolos de entrada y salida para proteger una sección crítica con la instrucción `fetch-and-increment`.

2. Implementa los protocolos de entrada y salida para proteger una sección crítica con la instrucción `compare-and-swap`.

3. Amplía los algoritmos de los protocolos de entrada y salida (también para el caso `test-and-set`) para que funcione con más de dos procesos garantizando espera limitada (Ayuda: usa un arreglo del tamaño del número de proceso participando para dar paso de forma explícita).

## 23.5. Estructuras de datos concurrentes

1. Implementa una lista concurrente. Ten los siguientes detalles en cuenta:

   - Queremos que varios hilos pueden insertar o borrar al mismo tiempo (claro, en diferentes sitios de la lista).
   - Desarrolla una semántica adecuado para una lista concurrente, entre otras, te debes contestar las siguientes preguntas:
     - ¿Debo mantener la longitud actual de la lista?
     - ¿Qué tipo de inserción uso como base para los demás?
     - ¿En qué orden adquiero los cerrojos necesarios?
     - ¿Cómo trato intentos concurrentes de borrar el mismo elemento al mismo tiempo?
     - ¿Qué hacen los iteradores?
     - ¿Cómo deben interactuar iteradores con los demás operaciones?
     - ¿Cómo implemento un caso de prueba?

2. Ayudas:
   - Busca en la red: Doug Lea (que ha hecho disponible una sugerencia para una librería de Java: `java.lang.concurrent`)
   - Mira como se ha implementado la cola.
   - Mira la solución del año pasado, cuando **ya has pensado cierto tiempo sobre una solución propia**.
3. Realiza la implementación usando las nuevas primitivas propuestas por la JSR166, es decir, implementa la lista con Java 1.5.
4. Aumenta la lista concurrente introduciendo también las operaciones con colecciones.

Tarea adicional para los interesados:

1. implementa una clase de lista concurrente en C++ con hilos `POSIX` o con hilos de la librería `glib`.
2. implementa una clase de lista concurrente en C# con hilos.

Para los que no quieren parar:

1. Implementa una tabla de dispersión concurrente (*hashtable*) aprovechando de la lista concurrente.
2. Implementa una tabla de dispersión concurrente (*hashtable*) con el método de *Cuckoo Hashing*.
3. Implementa un grafo concurrente (*graph*), a algunas de operaciones interesantes sobre dicha estructura de datos.
4. Implementa una estructura de datos que te resulta útil para desarrollar programas concurrentes.

# 24. Bibliografía

## 24.1. Básicas

1. D. Lea. *Programación Concurrente en Java.* Addison-Wesley, ISBN 84-7829-038-9, 2001.
2. J.T. Palma Méndez, M.C. Garrido Carrera, F. Sńchez Figueroa, A. Quesada Arencibia. *Programación Concurrente.* Thomson, ISBN 84-9732-184-7, 2003.
3. D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Pattern for Concurrent and Networked Objects.* John Wiley & Sons, ISBN 0-471-60695-2, 2000.
4. G. Coulouris, J. Dollimore, T. Kindberg. *Sistemas Distribuidos, Conceptos y Diseño.* Addison Wesley, ISBN 84-7829-049-4, 2001.

## 24.2. Complementarias

1. K. Arnold et.al. *The Java Programming Language.* Addison-Wesley, 3rd Edition, ISBN 0-201-70433-1, 2000.
2. B. Eckel. *Piensa en Java.* Prentice Hall, 2002.
3. M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, ISBN 0-13-711821-X, 1990.

## 24.3. Referencias adicionales

1. G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.
2. J.C. Baeten and W.P. Wiejland. *Process Algebra.* Cambridge University Press, 1990.
3. A. Burns and G. Davies. *Concurrent Programming.* Addison-Wesley, 1993.
4. C. Fencott. *Formal Methods for Concurrency.* Thomson Computer Press, 1996.
5. M. Henning, S. Vinoski. *Programación Avanzada en CORBA con C++.* Addison Wesley, ISBN 84-7829-048-6, 2001.
6. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
7. R. Milner. *Concurrency and Communication.* Prentice-Hall, 1989.
8. R. Milner. *Semantics of Concurrent Processes.* in J. van Leeuwen (ed.), Handbook of Theoretical Computer Science. Elsevier and MIT Press, 1990.
9. J.E. Pérez Martínez. *Programación Concurrente.* Editorial Rueda, ISBN 84-7207-059-X, 1990.
10. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

## 24.4. Enlaces en la Red

- Apuntes de esta asignatura:
  `http://www.ei.uvigo.es/~formella/doc/cd04/index.html`
- Concurrency JSR-166 Interest Site:
  `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`
- El antiguo paquete de Doug Lea que funciona con Java 1.4:
  `http://www.ei.uvigo.es/~formella/doc/concurrent.tar.gz` (.tar.gz [502749 Byte])
- The Java memory model:
  `http://www.cs.umd.edu/~pugh/java/memoryModel`