Concurrencia y Distribución

2008/2009

Dr. Arno Formella

Departamento de Informática
Universidad de Vigo

08/09

Concurrencia y Distribución I

Concurrencia y Distribución II

Concurrencia y Distribución III

Concurrencia y Distribución IV

Asignatura

Teoría:	los viernes, 12-14 horas, Aula 3.2
Prácticas:	3 grupos, los lunes 18-20 (SO5), los jueves 16-18 (31b), los viernes 10-12 (31b, en inglés)
Asignaturas vecinas:	todo sobre Programación, Sistemas Operativos, Procesamiento Paralelo, Redes, Sistemas en Tiempo Real, Diseño de Software
Prerrequisitos:	programación secuencial, sistemas operativos, algoritmos y estructuras de datos

Metodología

1. Clases magistrales donde se desarrollan los conceptos teóricos en pizarra y proyector
2. Clases en el laboratorio con herramientas y aplicaciones para ejercer los conocimientos adqueridos y usar los para realizar tareas de resolución de problemas y observación de propiedades de programas concurrentes
3. Lectura asignada para repetir concocimiento y adquerir pormedios propios nuevos aspectos
4. Realización de ejercicios con entrega de la documentación en pequeños grupos
5. Presentación y defensa oral de los resultados obtenidos de las tareas

Carga de trabajo

Actividades	**Horas Pres.**	**Factor**	**Horas NoPres.**	**TOTAL**
Clase magistral con avance teórico	28	0.5	14	42
Clase práctica en laboratorio en grupos	28	1	28	56
Coordinación en grupo, desarrollo de soluciones	0.5	15	7.5	8
Preparación de presentaciones	0.5	23	11.5	12
Preparación examen	0.5	9	4.5	5
Examen final	2	0	0	2
TOTAL	58		67	125

Evaluación

Evaluación:	80 % con un examen escrito al final del curso, teoría y práctica juntos 20 % por trabajos realizados durante las prácticas se puede obtener 25 % de los puntos del examen final (de junio) con presentaciones voluntarios durante las prácticas
Créditos:	6 (3 teoría, 3 prácticas)

Bibliografía I

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.

Bibliografía II

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.

Bibliografía III

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.

Bibliografía IV I

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.

Bibliografía V

- Apuntes de esta asignatura:
  `http://www.ei.uvigo.es/~formella/doc/cd06/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`

Bibliografía VI

- G. Bracha. *Generics in the Java Programming Language.* July 5, 2004.

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*

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`

Introducción

No existen definiciones muy claras de los terminos

- programación concurrente
- programación paralela
- programación distribuida

en la literatura.

Definición

Una posible distinción 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)

Definición

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.

Intuición

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

Sumamos

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 minutos

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo
- división del trabajo

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo
- división del trabajo
- distribución de los datos

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- fiabilidad de los componentes

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- fiabilidad de los componentes
- fiabilidad de la comunicación

Problemas

¿Con qué problemas nos enfrentamos?

- apuntamos:
- 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

Sumamos otra vez

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 minutos

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

Subdivisión

- Subdividimos una 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.

Procesos

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 (p.ej., en el acceso a los recursos, en la vista de memoria, en la priorisación etc.). En nuestro contexto no vamos a diferenciar mucho más.

Threads

- destacamos el concepto de hilo
  (se usa casi siempre en la programación moderna)
- un programa multi–hilo intercala varias secuencias de instrucciones que usan los mismos recursos (memoria común) bajo el techo de un sólo proceso
- (aquí proceso en el sentido de unidad de control del sistema operativo)
- el cambio de contexto de un hilo al siguiente dentro del procesador se suele realizar rápido
  (dentro de lo que cabe).

Resumen

- Un programa secuencial consiste en un sólo proceso.
- En un programa concurrente trabaja un conjunto de procesos en paralelo o cuasi paralelo.
- Los procesos cooperan para resolver un problema o realizar una tarea.
- La cooperación consiste especialmente en intercambiar información entre procesos.

Paralelismo virtual

- Los 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,
- p.ej., 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).

Entonces

La concurrencia describe un paralelismo potencial para la ejecución del programa en un sistema capaz de soportar lo.

Análisis

¿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

Indicadores I

¿Cuáles son indicadores que sugieren un programa concurrente?

- el problema consiste de forma natural en gestionar eventos (asincronidad, “asynchronous programming”), sobre todo si se trata de sistemas en red con servicios implementados
- el problema consiste en proporcionar un alto nivel de disponibilidad, es decir, nuevos eventos recién llegados requieren una respuesta rápida (disponibilidad, “availability”)

Indicadores II

- 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”)

Indicadores III

- 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.

Recursos

Entre 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

Prácticas

El las prácticas solamente nos dedicaremos

- a la sincronización entre hilos
- y a estructuras de datos como recursos que varios hilos quieres usar a la vez.
- La distribución del trabajo a los procesadores quedará en manos del planificador de la máquina virtual de Java (MVJ) igual como el uso de la memoria en manos del recolector de memoria de la MVJ.

Ejemplos I

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

- sistemas operativos
  - soportar operaciones cuasi–paralelas
  - proveer servicios a varios usuarios a la vez (sistemas multi-usuario, sin largos tiempos de espera)
  - gestionar recursos compartidos (p.ej., 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

Ejemplos II

- 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

Ejemplos III

- el sistema tiene que estimar el futuro próximo sin perder la capacidad de reaccionar rápidamente a cambios bruscos

- sistemas de comunicación (p.ej., 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 (p.ej., documentos multimedia) y acción (p.ej., juegos sobre la red, juegos distribuidos)

Ejemplos IV

- 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, p.ej., incluyendo delegación de servicios

Programación moderna

- En particular, resultará escencial el desarrollo de un programa concurrente cuando la concurrencia de activades es un aspecto interno del problema a resolver.
- Programadores modernos tienen que saber cómo escribir programas que manejan multiples procesos.

Herramientas

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

Ejemplos

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

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.

Ventajas

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, p.ej. no dejar recursos disponibles sin uso durante mucho tiempo (p.ej. procesadores a disposición)
- y que el programa refleje o bien el modelo del problema real o bien la propia realidad

Desventajas I

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

Desventajas II

- 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

Desventajas III

- 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)

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.

Java

- Se destacan ciertas diferencias con C++ (otro lenguaje de programación orientado a objetos importante).
- Se comentan ciertos detalles del lenguaje que muchas veces no se perciben a primera vista.
- Se describen también las novedades de la versión 1.5 (o 5, depende del momento).
- Se introducen los conceptos ya instrínsicos de Java para la programación concurrente.

Hola mundo

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

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

¿Qué se comenta?

Existen tres posibilidades de escribir comentarios:

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

- se usa javadoc o doxygen para generar automáticamente la documentación
- se documenta lo que no es obvio y las interfaces
- es decir: respuestas al *¿Cómo?* y *¿Por qué?*

Objetos

- 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.

Clases

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.

Inicialización

- 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, C++, C#

- 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).
- Se resaltará algunos de las diferencias principales entre Java y C++.

hello world

```
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 vector de cadenas de letras (`String`).

Tipos

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

Modificadores de clases I

Se pueden 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

Modificadores de clases II

- 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`.

Tipos simples I

`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

Tipos simples II

- 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.
- En Java 5 la conversión de tipos simples a sus objetos correspondientes (y vice versa) es automático.
- 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.).

Enumeraciones I

- hasta Java 1.4 se realizó enumeraciones así:

```
public final int MONDAY=0;
public final int TUESDAY=1;
public final int ...;
```

- a partir de Java 5 también así:

```
enum Weekdays { MONDAY, TUESDAY, ... }
```

- `enum` es una clase y automáticamente `public`, `static` y `final` (vemos en seguida)
- tienen `toString()` y `valueOf()`

Enumeraciones II

- `enum` es una clases, es decir, se pueden añadir miembros y métodos
- ```
  enum Coin {
    UN(1), DOS(2), CINCO(5), ...
    private final int value;
    Coin(int value) { this.value=value; }
    public int value() { return value; }
  }
  ```
  ```

Enumeraciones III

- `values()` devuelve un vector de los tipos del enumerado
- los `enum` se pueden usar en `switch`

```
Coin coin=...;
switch(coin) {
  case UN:
  case DOS:
  ...
}
```

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 I

Modificadores de miembros siendo instancias de objetos:

- `final`: declara constantes si está delante de tipos simples (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

Modificadores de miembros II

- `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++)

Modificadores de miembros III

- `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 métodos I

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++)

Modificadores de métodos II

- `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)

Modificadores de métodos III

- 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.

Estructuras de control

Las estructuras de control son casi iguales a las de C++ (nota la extensión del `for` en Java 5):

- `if(cond) then block`
- `if(cond) then block else block`
- `while(cond) block`
- `do block while (cond);`
- `for(expr; expr; expr) block`
- `for(type var: array) block`
- `for(type var: collection) block`
- `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`.

Marcas I

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

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

Marcas II

- También existe un `continue` con marca que permite saltar al principio de un bucle más alla del actual.
- No existe el `goto` (pero 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`.

Operadores I

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++

Operadores II

- 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.

Palabras reservadas I

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
```

Palabras reservadas II

- 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.

Objetos y referencias a objetos

- No se pueden 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(...)
```

Construcción por defecto

- 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.

Constructores

- 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.

Inicialización estática

```
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;
  }
  ...
}
```

Inicialización cruzada

- 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.

Recolector de memoria

- 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`.
- Antes de ser destruido se exejuta el método `finalize()` del objeto (por defecto no hace nada).

Reinterpretación de tipos

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

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

Paso de parámetros I

- 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.

Paso de parámetros II

- Hasta Java 1.4 la lista de parámetros siempre era fija, no existía el concepto de listas de parámetros variables de C/C++.
- en Java 5 si existe tal posibilidad.
- 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.

Paso de parámetros III

- En Java 5 existen listas de parámetros variables
  `void func(int fixed, String... names) {...}`
- Los tres puntos `...` significan 0 o más parámetros.
- Solo el último parámetro puede ser variable.
- Se accede con el nuevo iterador `for`:
  `for(String name : names) {...}`

Parámetros no modificables no existen

- No se puede evitar posibles modificaciones de un parámetro (que sí se puede evitar en C++ declarando el parámetro como `const`-referencia).
- Declarando el parámetro como `final` solamente protege la propia referencia (paso por valor).
- Entonces, no se pueden cambiar los valores de variables de tipos simples llamando a métodos y pasarles como parámetros variables de tipos simples (como es posible en C++ con referencias).
- La declaración se puede usar como indicación al usuario que se pretende no cambiar el objeto (aunque el compilador no lo garantiza).

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).

Vectores

- Los vectores 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.

Control de acceso

- Java comprueba si los accesos a vectores 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.

Vectores son objetos

- Los vectores son objetos implícitos que siempre conocen sus propias longitudes (`values.length`) (diferencia con C++ donde un vector no es nada más que un puntero) y que se comportan como clases finales.
- No se pueden declarar los elementos de un vector como constantes (como es posible en C++), es decir, el contenido de los componentes siempre se puede modificar en un programa en Java.

`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.

Más sobre constructores

- 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 superclase con `super(...)` (ambos no exiten en C++).
- El constructor de la superclase 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.

Orden de construcción

La construcción de objetos sigue siempre el siguiente orden:

- construcción de la superclase, 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

Extender clases I

- 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 subclases o clases extendidas.
- Una subclase heredará todas las propriedades de la clase súperior, aunque se tiene solamente acceso directo a las partes de la superclase declaradas por lo menos `protected`.

Extender clases II

- 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 pueden sobreescribir métodos de la superclase.
- Si se ha sobreescrito una cierta función, las demás funciones con el mismo nombre (pero diferente signatura) siguen visibles desde la clase derivada (en C++ eso no es el caso).
- Dicho último aspecto puede provocar sorpresas... ¿Cuáles?

Acceso a métodos sobreescritos

- Si se quiere ejecutar dentro de un método sobreescrito el código de la superclase, 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 (diferencia a C++ donde `const` forma parte de la signatura).

Sobreescritura y excepciones

- 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 superclase.
- Dicho subconjunto puede ser el conjunto vacio.
- Si se llama a un método dentro de una jerarquia 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.

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).

Clases locales

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

La clase `Object` I

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)

La clase `Object` II

- `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.

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 solo puede tener declaraciones de objetos que son constantes (`final`) y estáticos (`static`).
- En otras palabras, todas las declaraciones de objetos dentro de interfaces automáticamente son finales y estáticos, aunque no se haya descrito explícitamente.

Interfaces y herencia

- 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 { ... }
```

Métodos de interfaces

- 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.

Implementación de interfaces

- 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.

Implementación

```
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
  }
}
```

Resumen: interfaces

Las interfaces se comportan como clases totalmente abstractas, es decir,

- no tienen miembros no–estáticos,
- nada diferente a público,
- y ningún código no–estático.

Tipos como variables

- Como ya existía en C++, se introdujo la posibilidad de usar tipos como variables en la definición de clases y métodos.
- Se realiza con una sintaxis parecida:
  `List<Animal> farm=new ArrayList<Animal>();`
- Con eso se evita las muchas transformaciones explícitas de tipos que antes su usaba sobre todo para agrupar objetos en colecciones.
- Es decir, se puede diseñar estructuras de datos sin especificar antemano con que tipo se trabrajá en concreto.
- Cuando se usa el compilador garantiza que el tipo concreto proporciona las propiedades necesarias.

Clases genéricas

```
class Something<T> {
  T something;
  public Something(T something) {
    this.something=something;
  }
  public void set(T something) {
    this.something=something;
  }
  public T get() {
    return something;
  }
}
```

Uso de clase genérica

- Usamos la clase `Something` con cadenas.
- Construcción:

```
Something<String> w= new Something<String>("word")
```

- Leer el “contenido”:

```
String s=w.get();
```

- Escribir el “contenido”:

```
w.set(new Double(10.0));
```

producerá un fallo de compilación, hay que usar una cadena como parámetro.

Métodos genéricos

```
class Anything {
  public <T> T get(T something) {
    return something;
  }
  public static <T> void write(T something) {
    out.println(something);
  }
}
```

Con métodos genéricos se pueden implementar funcionalidades que se quieren realizar con cualquier tipo de interés.

Polimorfismo paramétrico restringido

- Se puede declarar el tipo que se usa para especificar un tipo genérico asumiendo cierta herencia:

```
List<T extends Animal>
```

- Así en el uso del tipo `T` ya se sabe algo sobre sus funcionalidades (y el compilador lo comprueba).

Polimorfismo paramétrico anidado/encadenado

- Se puede expresar también que el tipo genérico se heredera de otro tipo genérico:

```
List<T extends Animal<E>>
```

- o que el tipo genérico ya viene dado por otro tipo genérico

```
LinkedList<LinkedList<T>>
```

Limitaciones del polimorfismo paramétrico

- No se puede instanciar un objeto de un tipo genérico, sino es dentro de una clase o método del mismo, es decir,
- `T e=new T();` está prohibido
- `List<T> L= new LinkedList<T>();` está permitido.

Polimorfismo paramétrico con comodín I

- Observa: `List<Object>` *no* es superclase de `List<String>`.
- Entonces, para escribir métodos (y clases) que trabajen con cualquier *tipo genérico* necesitamos una notación nueva:
- `List<?>`
- sirve para implementar por ejemplo

```
void write(List<?> L) {
  for(Object e : L) out.println(e);
}
```

Polimorfismo paramétrico con comodín II

- Los comodines adquieren forma en su construcción:

```
Collection<?> C = new ArrayList<String>();
```

- ahora la colección `C` contiene cadenas.
- Solo `null` se puede asignar a una variable del tipo comodín, siempre.
- Eso no funciona para pasar parámetros:

  ```
  <T> void add(Set<T> s, T t) {...}
  ```

  no se puede usar con un conjunto construido genéricamente

  ```
  add(new Set<String>(), new String("hi");
  ```

Polimorfismo paramétrico con comodín III

- Los comodines se pueden usar también para expresar la propia cadena de herencia que se quiere mantener:

```
Collection<? extends Shape> C
  = new ArrayList<Circle>();
```

- donde `Circle` tiene que ser un tipo cuya superclase es `Shape`.
- Dicho concepto se llama comodín limitado.
- Pero ya no existe la posibilidad de escribir (la relación no es reflexiva)

```
Collection<? extends Shape> C
  = new ArrayList<Shape>();
```

Polimorfismo paramétrico con comodín IV

- También se puede limitar el comodín desde abajo:

```
Collection<? super Circle> C
  = new ArrayList<Shape>();
```

- Aquí sí se puede escribir

```
Collection<? super Circle> C
  = new ArrayList<Circle>();
```

dado que la relación es reflexiva.

Polimorfismo paramétrico con comodín V

- Los tipos genéricos (tanto comodín o no-comodín) se transforman en tipos simples *antes* de la ejecución.
- Por eso no se tiene acceso a la variable del tipo, con la consecuencia que

```
List<String> S=new ArrayList<String>();
List<Integer> I=new ArrayList<Integer>();
out.println(S.getClass()==I.getClass());
```

  imprime `true`.
- Tampoco se puede averiguar el tipo, el siguiente código no compila:

```
Collection<String> S=new ArrayList<String>();
if(S instanceof Collection<String>) \{...\}
```

Polimorfismo paramétrico con comodín VI

- Hay que tomarse muy en serio posibles mensajes de aviso cuando se usa tipos genéricos y cambiar el código hasta que no aparezca ninguno.
- Sino, puede ocurrir una simple excepción de fallo en conversión de tipo en algún momento de la ejecución cuya razón será difícil de localizar.

try'n'catch

- Para facilitar la programación de casos excepcionales Java usa el concepto de lanzar excepciones.
- Una excepción es una clase predefinida y se accede con la sentencia

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

Orden de ejecución I

- El bloque `try` contiene el código normal por ejecutar.
- Un bloque `catch(ExceptionObject)` contiene el código excepcional por ejecutar en caso de que durante la ejecución del código normal (que contiene el bloque `try`) se produzca la excepción del tipo adecuado.
- Pueden existir más de un (o ningún) bloque `catch` para reaccionar directamente a más de un (ningún) tipo de excepción.
- Hay que tener cuidado en ordenar las excepciones correctamente, es decir, las más específicas antes de las más generales.

Orden de ejecución II

- El bloque `finally` se ejecuta siempre una vez terminado o bien el bloque `try` o bien un bloque `catch` o bien una excepción no tratada o bien antes de seguir un `break`, un `continue` o un `return` hacia fuera de la sentencia `try-catch-finally`.

Construcción de clases de excepción

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

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

Declaración de excepciones lanzables

- Entonces, una excepción no es nada más que un objeto que se crea en el caso de aparición del caso excepcional.
- La clase principal de una excepción es la interfaz `Throwable` que incluye un `String` para mostrar una línea de error legíble.
- Para que un método pueda lanzar excepciones con las sentencias `try-catch-finally`, es imprecindible declarar las excepciones posibles antes del bloque de código del método con `throws ...`.

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

- En C++ es al revés, se declara lo que se puede lanzar como mucho.

Propagación de excepciones

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

Lanzar excepciones

- Se pueden lanzar excepciones directamente con la palabra `throw` y la creación de un nuevo objeto de excepción, por ejemplo:

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

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

Excepciones de ejecución

- Además de las excepciones así declaradas existen siempre excepciones que pueden ocurrir en qualquier momento de la ejecución del programa, por ejemplo, `RuntimeException` o `Error` o `IndexOutOfBoundException`.
- La ocurrencia de dichas excepciones refleja normalmente un flujo de control erróneo del programa que se debe corrigir antes de distribuir el programa a posibles usuarios.
- Se usan excepciones solamente para casos excepcionales, es decir, si pasa algo no esperado.

Agrupación de objetos I

- Siempre existe la posibilidad de que diferentes fuentes usen el mismo nombre para una clase.
- Para producir nombres únicos se agrupa los objetos en paquetes.
- El nombre del paquete sirve como prefijo del nombre de la clase con la consecuencia de que cuando se diferencian los nombres de los paquetes también se diferencian los nombres de las clases.

Agrupación de objetos II

- Por convención se usa como prefijo el dominio en internet en orden inverso para los paquetes.
- Hay que tener cuidado en distinguir los puntos en el nombre del paquete con los puntos que separan los miembros de una clase.
- La pertenencia de una clase a un paquete se indica en la primera sentencia de un fichero fuente con

```
package Pack.Name;
```

¿Por qué le gusta Java tanto a la gente?

- Java viene con una amplia gama de clases y paquetes predefinidos, por ejemplo, `AWT`, `Swing`.
- Se accede a los paquetes importándolo necesario con `import`.
- Se accede a los componentes de los paquetes con clasificadores, p.ej., `System.out.println(...)` (en Java 5 ya no hace falta clasificar, se importa como `import static java.lang.System.*`).
- Cuidado: Java no está disponible siempre en todas las plataformas en su última versión y eso puede derivar en aplicaciones no portables.

Clase de cadenas de caracteres

- Java proporciona la clase `String` (cadenas) con muchos métodos ya implementados. Si se requiere muchas operaciones de cadenas que modifican el contenido de la cadena, mejor usar la clase `StringBuffer`.
- Eso es justamente una gran potencia de Java: la disponibilidad de un gran conjunto de paquetes ya implementado (para C++ hay que buscar un poco más).

Acceso a si mismo

- Java proporciona para cada clase un objeto de tipo `Class` que se puede usar para obtener información sobre la propia clase y todos sus miembros.
- Así por ejemplo se puede averiguar todos los métodos y modificadores, cual es su clase súperior y mucho más.

Objetivos

Se usan los hilos para ejecutar varias secuencias de instrucciones de modo cuasi–paralelo.

Creación de un hilo (para empezar)

- Se crea un hilo con

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

- Después se inicializa el hilo y se define su comportamiento.
  Se lanza el hilo con

```
worker.start()
```

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

La interfaz `Runnable`

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

PingPONG I

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

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

PingPONG II

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

PingPONG III

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

Construcción de `Runnable`s

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

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

Implementación de `Runnable`

- La interfaz `Runnable` exige solamente el método `run()`, sin embargo, normalmente se implementan más métodos para crear un servicio completo que este hilo debe cumplir.
- Aunque no hemos guardado las referencias de los hilos en unas variables, los hilos *no caen* en las manos del recolector de memoria: siempre se mantiene una referencia al hilo en su grupo al cual pertenece.
- El método `run()` es público y en muchos casos, implementando algún tipo de servicio, no se quiere dar permiso a otros ejecutar directamente el método `run()`. Para evitar eso se puede recurrir a la siguiente construcción:

`run()` no-público

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

Explicación del ejemplo

- Crear el servicio con `Service()` lanza un nuevo hilo que actua sobre una cola para realizar su trabajo con cada tarea que encuentra ahí.
- El trabajo por hacer se encuentra en el método privado `realService()`.
- Una nueva tarea se puede añadir a la cola con `AddJob(...)`.
- **Nota:** la construcción arriba usa el concepto de clases anónimas de Java, es decir, sabiendo que no se va a usar la clase en otro sitio que no sea que en su punto de construcción, se declara directamente donde se usa.

Sincronización

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

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

- La expresión entre paréntesis `obj` tiene que evaluar a una referencia a un objeto o a un vector.
- Declarando un método con el modificador `synchronized` garantiza que dicho método se ejecuta ininterrumpidamente por un sólo hilo.
- La máquina virtual instala un cerrojo (mejor dicho, un monitor, ya veremos dicho concepto más adelante) que se cierra de forma atómica antes de entrar en la región crítica y que se abre antes de salir.

Métodos syncronizados

- Declarar un método como

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

  es equivalente a usar un bloque sincronizado en su interior:

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

- Los monitores permiten que el mismo hilo puede acceder a otros métodos o bloques sincronizados del mismo objeto sin problema.
- Se libera el cerrojo sea el modo que sea que termine el método.
- Los constructores no se pueden declarar `synchronized`.

Syncronización y herencia

- No hace falta mantener el modo sincronizado sobreescribiendo métodos síncronos mientras se extiende una clase. (No se puede *forzar* un método sincronizada en una interfaz.)
- Sin embargo, una llamada al método de la clase súperior (con `super.`) sigue funcionando de modo síncrono.
- Los métodos estáticos también pueden ser declarados `synchronized` garantizando su ejecución de manera exclusiva entre varios hilos.

Protección de miembros estáticos

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

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

¡Ojo con el concepto!

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

Objetos síncronos

Se obtienen objetos totalmente sincronizados siguiendo las reglas:

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

Páginas del manual

Se recomienda estudiar detenidamente las páginas del manual de Java que estén relacionados con el concepto de hilo.

Atomicidad en Java

- Solo las asignaciónes a variables de tipos simples de 32 bits son atómicas.
- `long` y `double` no son simples en este contexto porque son de 64 bits, hay que declararlas `volatile` para obtener acceso atómico.

Limitaciones para la programación concurrente

- no se puede interrumpir la espera a un cerrojo (una vez llegado a un `synchronized` no hay vuelta atrás)
- no se puede influir mucho en la política del cerrojo (distinguir entre lectores y escritores, diferentes justicias, etc.)
- no se puede confinar el uso de los cerrojos (en cualquier línea se puede escribir un bloque sincronizado de cualquier objeto)
- no se puede adquirir/liberar un cerrojo en diferentes sitios, se está obligado a un estructura de bloques

Paquete especial para la programación concurrente

- Por eso se ha introducido en Java 5 un paquete especial para la programación concurrente.

  `java.util.concurrent`

- Hay que leer todo su manual.

Java: seguridad y compatibilidad

- Muchas veces se oye que Java es un lenguaje seguro porque es tan estricto con sus tipos y la máquina virtual de Java puede prohibir ciertas acciones (como, por ejemplo, escribir en el disco o acceder a ciertos recursos).
- Sin embargo, hay que tener en cuenta que Java no es más seguro que la implementación de la máquina virtual.
- Java solo es tan compatible como son sus versiónes referiéndose a las implementaciones de las máquinas virtuales y de los paquetes en las diferentes plataformas.

Entonces ¿Qué es un hilo?

- Un hilo es una secuencia de instrucciones
  - que está controlada por un planificador y
  - 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.

Paquete de hilos

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();
```

Ejemplo: el ping

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 ");
    }
  }
}
```

Ejemplo: el PONG

El hilo hereda todo de la clase `Thread`, pero sobreescribe 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 ");
    }
  }
}
```

Programa/hilo principal

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

Resultados

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

Objetivo

Nuestro objetivo es:

la ejecución del pingPONG independientemente del sistema debajo y que siempre funciona bien.

Intento con dormir I

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;
      }
    }
  }
}
```

Intento con dormir II

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

Resultados

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

Intento con ceder I

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();
    }
  }
}
```

Intento con ceder II

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

Resultados

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

Solución

Prácticas: codificar los ejemplos y realizar un pingPONG perfecto.

Algoritmo secuencial

- Asumimos que tengamos solamente las operaciones aritméticas *sumar* y *subtraer* disponibles en un procesador fictício y queremos multiplicar dos números positivos.

Algoritmo 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:  set p and q to positive numbers

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 sí 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:`)

¿Cómo se comprueba sí 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,

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

- 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.

- 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.

- 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 corrección hay que comprobar con el concepto de inducción es entonces:

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

Algoritmo concurrente

Reescribimos el algoritmo secuencial para que “funcione” con dos procesos:

```
Initially:  set p and q to positive numbers

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: ...
```

Intercalaciones posibles

- El algoritmo no es determinista,
- 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 nodeterminismo 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)

Funcionamiento correcto I

Generalmente se dice que un programa es correcto si dada una entrada el programa produce los resultados deseados. Más formal:

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

Funcionamiento correcto I

Generalmente se dice que un programa es correcto si dada una entrada el programa produce los resultados deseados.
Más formal:

- 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.

Funcionamiento correcto II

Se define dos tipos de funcionamiento correcto de un programa:

funcionamiento correcto parcial:

dada 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 II

Se define dos tipos de funcionamiento correcto de un programa:

funcionamiento correcto parcial:

dada 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.

Funcionamiento correcto III

- Un ejemplo es el cálculo de la raíz cuadrada, 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`.
- (Entonces para `nan` como argumento también hay que definir todas las funciones.)

Funcionamiento correcto IV

- 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.

Funcionamiento correcto V

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.

Funcionamiento correcto VI

Entonces:

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.

Funcionamiento correcto VI

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 por parte del planificador.

Funcionamiento correcto VII

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

Funcionamiento correcto VII

- 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.

Imposibilidad de comprobación exhaustiva

- 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 era 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.

Dependencia de atomicidad I

```
P1:   inc N
P2:   inc N

P2:   inc N
P1:   inc N
```

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

Dependencia de atomicidad II

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

¿Qué es exclusión mutua?

- 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.
- Es importante entender cómo se implementan los protocolos de entrada y salida para realizar la exclusión mutua.
- Obviamente no se puede implementar exclusión mutua usando exclusión mutua: se necesita algo más básico.
- Un método es usar un tipo de protocolo de comunicación basado en las instrucciones básicas disponibles.

Estructura general basada en protocolos

Entonces el protocolo para cada uno de los participantes refleja una estructura como sigue:

```
P0                          ... Pi
...\\                           ...\\
entrance protocol               entrance protocol
critical section                critical section
exit protocol                   exit protocol
...\\                           ...\\
```

Un posible protocolo (asimétrico)

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

Principio de la bandera

Si

- ambos procesos primero levantan sus banderas
- y después miran al otro lado

por lo menos un proceso ve la bandera del otro levantado.

Comprobación con contradicción

- asumimos P0 era el último en mirar
- entonces la bandera de P0 está levantada
- asumimos que P0 no ha visto la bandera de P1
- entonces P1 ha levantado la bandera después de la mirada de P0
- pero P1 mira después de haber levantado la bandera
- entonces P0 no era el último en mirar

Propiedades de interés del protocolo

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

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

Obviamente se asume que ningún proceso ocupa la sección crítica durante un tiempo infinito.

Propiedades de interés del protocolo

La propiedad de espera finita se puede analizar según los siguientes criterios:

justicia:
: hasta que medida influyen las peticiones de los demás procesos en el tiempo de espera de un proceso

espera:
: hasta que medida influyen los protocolos de los demás procesos en el tiempo de espera de un proceso

tolerancia a fallos:
: hasta que medida influyen posibles errores de los demás procesos en el tiempo de espera de un proceso.

Análisis del protocolo asimétrico

Analizamos el protocolo de antes respecto a dichos criterios:

- ¿Está garantizado la exclusión mutua?
- ¿Influye el estado de uno (sin acceso) en el acceso del otro?
- ¿Quién gana en caso de peticiones simultaneas?
- ¿Qué pasa en caso de error?

Soporte hardware

- Dependiendo de las capacidades del hardware la implementación de los protocolos de entrada y salida es más facil o más difícil, además las soluciones pueden ser más o menos eficientes.
- 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 básicas que permiten realizar los protocolos de forma más eficiente.

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
```

Primer intento: propiedades

- 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.

Segundo intento

Intentamos evitar la alternancia. Usamos para cada proceso una variable, `v0` para `P0` y `v1` para `P1` respectivamente, que indican 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
```

Segundo intento: propiedades

- 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?

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
```

Tercer intento: propiedades

- 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?

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
```

Cuarto intento: propiedades

- Está garantizado la exclusión mutua.
- Se puede producir una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil (*livelock*)

- ¿Cuál es la intercalación maligna?

Algoritmo de Dekker: quinto intento

```
Initially:  v0,v1 are equal to false, v is equal to 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
```

Quinto intento: propiedades

El algoritmo de Dekker resuelve el problema de exclusión mutua en el caso de dos procesos, donde se asume que la lectura y la escritura de un valor íntegro de un registro se puede realizar de forma atómica.

Algoritmo de Peterson

```
  P0                          P1
a: loop                        loop
b:  set v0 to true              set v1 to true
c:  set v to P0                 set v to P1
d:  wait while                  wait while
e:   v1 equals true              v0 equals ture
f:   and v equals P0             and v equals P1
g:  critical section            critical section
h:  set v0 to false             set v1 to false
i:  non-critical section        non-critical section
j: endloop                     endloop
```

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 entre todos los procesos esperando
- el proceso con el valor mínimo accede la sección crítica

Algoritmo de Lamport: observaciones

- se necesita un cerrojo (acceso con exclusión mutua) 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

El algoritmo no es verdaderamente practicable dado que se necesitan infinitos tickets y un número elevado de comprobaciones.

Otros algoritmos

- 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.
- Existe una generalización del algoritmo de Peterson para *n* procesos (filter algorithm).
- Se puede evitar la necesidad de un número infinito de tickets, si se conoce antemano el número máximo de participantes (uso de grafos de precedencia).
- Otra posibilidad es al algoritmo de Eisenberg–McGuire. (mirad la bibliografía).

Límites

- Se puede comprobar que se necesita por lo menos n campos en la memoria para implementar un algoritmo (con `load` and `store`) que garantiza la exclusión mutua entre n procesos.

Operaciones en la memoria

- 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.

TAS

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.

TAS

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

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

TAS: propiedades

- 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).
- ¿Cómo se puede garantizar una espera limitada?

EXCH

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.

EXCH

```
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
```

EXCH: propiedades

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

F&A

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?

CAS

- 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?

double CAS

Existe también una mejora del `CAS`, llamado *double-compare-and-swap*, 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
    swap S1 and V1
    swap S2 and V2
    return true
  else
    return false
```

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 indica 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”).

Propiedades de seguridad

Las propiedades de seguridad suelen ser algunas de las invariantes del programa que se tienen que introducir en las comprobaciones del funcionamiento correcto.

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 ninguna situación en la cual todos los procesos participantes quedan atrapados en una espera a una condición que nunca se cumpla.

Propiedades de vivacidad I

Inanición:

- Un proceso puede “morirse” por inanición (“starvation”), es decir, un proceso o varios procesos siguen con su trabajo pero otros nunca avanzan por ser excluidos de la competición por los recursos (por ejemplo en Java el uso de `suspend()` y `resume()` no está recomendado por esa razón).
- Existen problemas donde la inanición no es un problema real o es muy improbable que ocurra, es decir, se puede aflojar las condiciones a los protocolos de entrada y salida.

Propiedades de vivacidad II

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 por esa razón).

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

Justicia entre procesos

Cuando los procesos compiten por el acceso a recursos compartidos se pueden 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.

Comentarios

- Los dos primeros conceptos son conceptos teóricos porque dependen de términos *infinitamente* o *en 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.
- Normalmente se quiere que todos los procesos manifesten algún progreso en su trabajo (pero en algunos casos aninción controlada puede ser tolerada).

Espera activa de procesos

- El algoritmo de Dekker y sus parecidos 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.

Evitar epera infinita o inanición de procesos

- Se implementa el acceso a recursos compartidos siguiendo un 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.

Conceptos

- 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 anteriormente.

Semáforo

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso de manera exclusiva cuando varios procesos están compitiendo y que 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 debe llegar a ser negativo.
- Existen tres operaciones con un semáforo: `init()`, `wait()`, y `signal()` que realizan lo siguiente:

Operaciones del semáforo I

`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.

Operaciones del semáforo II

`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.

Operaciones del semáforo III

`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 ha despertado un proceso con exito en caso que había alguno bloqueado.
- Para despertar los procesos se pueden implementar varias formas que se destinguen en su política de justicia (p.ej. FIFO).

Uso del semáforo

El acceso mutuo a secciones 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 section                 critical section
d:    S.signal()                      S.signal()
e:    non-critical section             non-critical section
f: endloop                          endloop
```

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

Semáforos binarios/generales

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.

Detalles de la implementación I

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

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

Detalles de la implementación II

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

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

Principales desventajas de semáforos

- No se puede imponer el uso correcto de las llamadas a 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.

Región crítica

- Un lenguaje de programación puede realizar directamente una implementación de una región crítica.
- Así parte de la responsabilidad se traslada 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
```

Observaciones

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

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, vistas hasta ahora, 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
```

Detalles de implementación I

Las regiones críticas condicionales funcionan internamente de la siguiente manera:

- Un proceso que quiere entrar en la región crítica espera hasta que tenga permiso.
- 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.

Detalles de implementación II

- 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.

Detalles de implementación III

- Nota que esta técnica puede derivar en muchas comprobaciones de la condición, todos en modo exclusivo, y puede causar pérdidas 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.

Desventajas de semáforos y regiones críticas

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 variables globales.

Monitor

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.

Detalles de implementación de un monitor

- El acceso al monitor está permitido solamente a través de los métodos públicos y el compilador garantiza exclusión mutua para todos los accesos.
- 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.

Sincronización condicional

- 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()`/`notifyAll()`).
- 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.

Disponibilidad de monitores

- Lenguajes de alto nivel que facilitan la programación concurrente suelen tener monitores implementados dentro del lenguaje (por ejemplo en Java).
- El uso de monitores es bastante costoso, porque se pierde eficiencia por bloquear mucho los procesos.
- Por eso se intenta aprovechar de primitivas más potentes para aliviar este problema.

Desventajas de uso de sincronización a alto nivel

- No se distingue entre accesos de solo lectura y de escritura que limita la posibilidad de accesos en paralelo.
- Cualquier interrupción (p.ej. por falta de página de memoria) relantiza el avanze de la aplicación.
- Por eso las MVJ usan los procesos del sistema operativo para implementar los hilos, así el S.O. puede conmutar a otro hilo.

Problema clásico

El problema del productor y consumidor es un ejemplo clásico de programa concurrente y consiste en la situación siguiente: 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:
  forever
    produce(item)
    place(item)
```

```
consumer:
  forever
    take(item)
    consume(item)
```

Requerimientos

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, pero no debe producir nada si no lo puede colocar
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

Cola infinita

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 que garantiza exclusión mutua a la cola:

```
producer:
  forever
    produce(item)
    place(item)
    itemsReady.signal()
```

```
consumer:
  forever
    itemsReady.wait()
    take(item)
    consume(item)
```

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

Corrección

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.

Más participantes

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:
  forever
    produce(item)
    mutex.wait()
    place(item)
    mutex.signal()
    itemsReady.signal()
```

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

Cola finita

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.

Cola finita

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

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

Granularidad

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

Clases de architecturas

Una clasificación clásica de ordenadores paralelos es:

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

Procesadores modernos

- 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.

graphics processing units (GPU)

La programación paralela y concurrente (y con pipeline) se revive actualmente en la programación de las GPUs (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.
Los procesadores sueles usar solamente precisión simple.

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).

Computación en red

- La visión de ‘computación en la red’ no es nada más que un gran sistema MIMD.
- Existe una nueva tendencia de usar un llamado GRID de superordenadores para resolver problemas grandes (y distribuir el uso de los superordenadores entre más usuarios).

Mecanismos de conmutación

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.

Mecanismos de conmutación

- Al desarrollar un programa concurrente, no se debe asumir ningún comportamiento específico del planificador (siendo la unidad que realiza la conmutación de los procesos).
- No obstante, un planificador puede analizar los programas concurrentes durante el tiempo de ejecución para adaptar el mecanismo de conmutación hacia un mejor rendimiento (ejemplo: automatic “nice” en un sistema Unix).
- También los sistemas suelen ofrecer unos parámetros de control para influir en las prioridades de los procesos que se usa como un dato más para la conmutación.

Memoria compartida homogéneo (SMP)

- Sin una memoria compartida no existe concurrencia (se necesita por lo menos unos registros con acceso común).
- 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)
- 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.

Memoria compartida heterogéneo

- 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).

Comunicación y sincronización

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.

Sistemas híbridos

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.

Comunicación

Programas concurrentes o/y distribuidos 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.

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).

Síncrono y asíncrono

- 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.

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)
- Muchas veces se realiza los flujos con una comunicación con paquetes añadiendo capas de control (pila de control).

Posibles fallos en canales de paquetes

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

Técnicas para superar los problemas

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

Comunicación segura sin mensajes de recibo

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*).

Problema

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.

Condiciones necesarias

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

Funcionamiento parcial

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).

Técnicas de actuar

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

- Detectar y actuar
- Evitar
- Prevenir

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.

Actuación

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.

Desventaja

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.

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. después 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.

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

Prevenir exclusión mutua

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).

Prevenir accesos consecutivos

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

Prevenir uso único

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*
- (Separar lectores y escritores alivia este problema también.)

Prevenir ciclos

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.

Problematica específica

- terminación distribuida
- gestión de memoria distribuida
- estado distribuido
- propiedades distribuidos
- tiempo distribuido
- comunicación

Paso de mensajes
distribución física y lógica

- Un proceso manda un mensaje a otro proceso (que suele esperar dicho mensaje).
- Es imprecindible en sistemas distribuidos (no existen recursos directamente compartidos para intercambiar información entre procesos)
- También si se trabaja con un solo procesador pasar mensajes entre procesos es un buen método de sincronizar procesos y/o trabajos.
- Existen muchas variantes de implementaciones para el paso de mensajes.

Destacamos unas características.

Tipos de sincronización

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

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

Espera finita

- si antemano se desconoce el tiempo de paso de mensajes
- Los remitentes y los receptores pueden implementar una espera finita (con temporizadores) para no quedar bloqueado eternamente al no llegar información necesaria del otro lado.
- Se necesita un mécanismo de vigilancia del canal, o bien por interrupción o bien por inspección periódica.
- Sobre todo por razones de eficiencia es conveniente distinguir entre mensajes locales y mensajes a procesadores remotos.

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

Prioridades

- Para el paso de mensajes se 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`).

Terminación de programas

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

- terminar con éxito
- terminar con excepción
- terminar con interrupción
- terminar nunca a propósito
- terminar nunca por fallo

Terminación de programas concurrentes

- 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 es un sistema distribuido).
- Un programa concurrente termina cuando todas sus partes secuenciales han terminado.

Terminación en programas distribuidos

- 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.

Detección de la 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.

Invariantes por mantener

- 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.

Algoritmo de detección de terminación I(V)

- 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.

Algoritmo de detección de terminación II(V)

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

Algoritmo de detección de terminación III(V)

Ampliamos el algoritmo para que funcione también con grafos acíclicos.

- Añadimos a cada arista un campo “déficit” que se aumenta siempre que se ha 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.

Algoritmo de detección de terminación IV(V)

- 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, se observa que forman un árbol abarcador (*spanning tree*) del grafo.

Algoritmo de detección de terminación V(V)

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.

Sincronización del tiempo
problemática

¿Cómo conseguir que los procesos en procesadores distribuidos tengan la misma noción del tiempo?

Sincronización del tiempo
reloj sincronizado

- Se implementa un reloj centralizado (hardware) con el cual todos los procesadores se sincronizan con una desviación casi no apreciable.
- Un sistema implementado así es el GPS donde todos los satelites tienen un reloj atómico altamente sincronizado.
- Se consigue una sincronización en fracciones de nanosegundos.

Sincronización del tiempo
reloj en red

- Se implementa un protocolo con un servidor de tiempo (NTP, network time protocol) con el cual los procesadores se sincronizan de vez en cuando.
- No se puede ajustar el reloj hacia detrás (produciría p.e. ficheros creados en el futuro).
- Se deja *transcurrir* el tiempo más rápido o más lento (modificando las interrupciones periódicas).
- Se implementa una jerarquia de servidores donde los mejores (relojes atómicos) nunca se modifican, sino lo hacen los demás y siempre el de menos precisión con el de más, y en el caso de empate mutuamente.
- Existen los protocolos con servidor activo y pasivo.

Sincronización del tiempo
reloj lógico

- Se sincroniza los procesos con los sellos de tiempo que se transmite junto con todos los mensajes.
- Se observa: dos eventos en el mismo procesor son fáciles de ordenar en el tiempo.
- Se asume (obviamente): el paso de mensajes, como mucho, consume tiempo.
- Cada proceso ajusta su reloj hacia delante (en una de la capas centrales del modelo OSI) cada vez que recibe un mensaje.
- Se puede implementar una ordenación global con acusos de recibi y ordenación de los mensajes en colas (en una capa media) que transmiten las cabeceras a la aplicación cuando no falta ningún recibo.

Sincronización del tiempo
reloj vectorial

- Los relojes lógicos solamente ordenan los mensajes en tiempo, no por causalidad.
- Idea: se mantiene un vector en cada proceso que contiene todos los relojes lógicos de todos los procesos involucrados (de hecho basta con almacenar el número de eventos de sincronización).
- Un proceso puede ordenar los eventos según su causalidad porque tiene acceso a todos los ordenaciones.
- No se puede distinguir entre causalidad temporal y causalidad lógico.

Concepto

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

Matemática

técnicas de pruebas matemáticas:

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

Algoritmia

paradigmas de desarrollo de algoritmos:

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

Patrones

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

- Reactor
- Proactor
- Ficha de terminación asíncrona
- Aceptar–Conector
- Guardián
- Interfaz segura para multi–hilo
- Aviso de hecho
- Objetos activos
- Monitor
- Mitad–síncrono/mitad–asíncrono
- Líder–y–Seguidores

Uso

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 (hay que usar programaciÃşn mulit–hilo con sincronización).

Uso

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.

Uso

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 una 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.

Uso

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 I

- 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.

Detalles de la implementación II

- 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.

Uso

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 I

- 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.

Detalles de la implementación II

- 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*).

Uso

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:

- uso de objetos compartidos

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.

Uso

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:

- construcción de singletons

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.
- Se quiere inicializar solamente en caso de necesidad real.

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.

Uso

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.

Uso

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).

Uso

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

Uso

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

Comentarios

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

Empezando I

1. Consigue el “Hola Mundo” en Java.
2. Consigue un “Hola Mundo, soy hilo ...” usando varios hilos (mira detenidamente en los manuales de Java la clase `Thread` y también la interfaz `Runnable`).
3. Mide cuantos hilos se puede lanzar y mantener vivos simultaneamente.
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. Realiza un diagrama dibujando tiempo de ejecución frente a números de hilos.

Empezando II

5. Cambia el trabajo que realiza un hilo (escribir a consola) por algo que no tenga salida, observa las diferencias comparándolo con los resultados de antes (realiza los mismos gráficos que arriba).
6. Asegúrate que tu programa termina bien, es decir, que todos los hilos participantes lleguen a su último "}".

Describe de forma precisa todos tus observaciones (dependiendo del tipo de S.O., de la ocupación del ordenador, del trabajo por realizar, etc.).

PingPONG I

1. Implementa un `pingPONG` perfecto. Ten los siguientes detalles en cuenta:
   - Experimenta con los diferentes intentos presentados en los apuntes.
   - Desarrolla una solución con las siguientes propiedades:
     - Usa tres hilos (un hilo para el programa principal, o el árbitro, y un hilo para cada jugador).
     - El árbitro inicia el juego (mensaje previo a la pantalla).
     - Los jugadores producen sus `ping`s y `PONG`s alternamente.
     - El árbitro termina el juego después de cierto tiempo (mensaje previo a la pantalla).
     - Los jugadores realizen como mucho un intercambio de pelota más.

PingPONG II

- Ambos jugadores/hilos terminan (mensaje previo a la pantalla).
- El árbitro escribe el último mensaje.
- El programa termina.

- Observa la diferencia entre el uso de `notify()` y `notifyAll()`, sobre todo respecto a despertas "inútiles" de hilos.

2. 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`).

PingPONG III

3. ¿Cuál sería una implementación perfecta? (es decir una implementación en la cual se despierta solamente al hilo que tiene que jugar en este instante).
4. Implementa el `pingPONG` entre dos ordenadores.
   - Asume que los IPs estén conocidos antemano.
   - Duplica la salida en los tres ordenadores, cada uno pone un prefijo delante, por ejemplo, `arbitro:`, `jugador rojo:`, y `jugador azul`.

Planificación con prioridades I

Implementa 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 en los últimos k accesos por lo menos unos a% de los accesos son para los procesos de tipo A, b% para los del tipo B y c% para los del tipo C? Obviamente, los percentages

Planificación con prioridades II

valen solamente si hay procesos de tal tipo esperando en tal momento y su suma no puede superar los 100 %. (Ayuda: un planificador sabe contar).

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

Estructuras de datos concurrentes I

1. Preparación:
   - Estudia detenidamente el paquete `java.util.concurrent`.
   - Estudia la implementación de una lista concurrente `http://trevinca.ei.uvigo.es/~formella/doc/cd06/ConcurrentList.tgz`.
2. Usa la lista concurrente para implementar una tabla de dispersión (hashtable) de la siguiente manera:
   - Existe una array de tamaño fijo que contiene para cada clave (campo en el array) una lista concurrente que almancena a su vez las entradas con dicha clave.

Estructuras de datos concurrentes II

- Implementa por lo menos las funcionalidades: `insert` (se inserte un nuevo objeto en la tabla), `lookup` (se devuelve verdadero si el objeto se encuentra en la tabla, sino falso), y `delete` (se borra un objeto, si existe en

3. Implementa un caso de uso de dicha tabla de dispersión lo suficientemente grande para realizar mediciones de tiempo de ejecución.
4. Compara tu implementación con la `ConcurrentHashMap` de Java respecto a tiempo de ejecución y uso de memoria.