Concurrencia y Distribución

2012/2013

Tercero, Grado

Dr. Arno Formella

Departamento de Informática
Universidade de Vigo

12/13

profesorado

Profesor:	Arno FORMELLA
Web:	`http://trevinca.ei.uvigo.es/~formella`
Correo:	formella@uvigo.es (formella@ei.uvigo.es)
Tutorías	Ma: 09:30-13:30, 17:00-18:00
	Mi: 12:00-13:00

Profesor:	Emilio GARCÍA ROSELLO
Web:	usa TEMA
Correo:	erosello@uvigo.es
Tutorías	Lu: 09:30-10:30, 14:30-16:30
	Ju: 09:30-10:30

tutorías

- Cambios puntuales de tutorías via aviso web.
- Idiomas: galego, castellano, English, Deutsch.

horas de dedicación (según guía)

	pres.	no-pres.	suma
Actividades introductorias	1.5	–	1.5
Sesión magistral	18.0	9.0	27.0
Estudios/actividades previos	–	16.0	16.0
Prácticas en aulas de informática	26.5	26.5	53.0
Resolución de problemas y/o exercicios	–	19.5	19.5
Presentacións/exposicións	–	1.75	1.75
Tutoría en grupo	1.25	1.25	2.5
Pruebas de respuesta corta	1.5	–	1.5
Pruebas de respuesta larga	2.0	–	2.0
Informes/memorias de prácticas	–	12.0	12.0
Probas prácticas	1.0	–	1.0
Resolución de problemas y/o exercicios	–	12.0	12.0
Otras	0.25	–	0.25
Suma	52.0	98.0	150.0

horas presenciales

Teoría:	los viernes, 9:00-10:30 horas, Aula 2.2
Prácticas: CDI_1/2 CDI_3/4	4 grupos, Lab. 31a los jueves 12:30-14:30/10:30-12:30 los lunes 10:30-12:30/12:30-14:30
Prerrequisitos:	matemáticas, algoritmos y estructura de datos, programación, arquitectura de computadoras, redes, sistemas operativos, lenguajes de programación

contexto en el plan de estudio

ORGANIZACIÓN TEMPORAL DEL TÍTULO DE GRADUADO/A EN INGENIERÍA INFORMÁTICA

SEMESTRES							
1/1S	1/2S	2/1S	2/2S	3/1S	3/2S	4/1S	4/2S
DERECHO:: FUNDAMENTOS ÉTICOS Y JURÍDICOS DE LAS TIC (CFB; 6 ECTS)	EMPRESA:: ADMINISTRACIÓN DE LA TECNOLOGÍA Y LA EMPRESA (CFB; 6 ECTS)	INGENIERÍA DEL SOFTWARE I (OB; 6 ECTS)	INGENIERÍA DEL SOFTWARE II (OB; 6 ECTS)	INTERFACES DE USUARIO (OB; 6 ECTS)	DIRECCIÓN Y GESTIÓN DE PROYECTOS (OB; 6 ECTS)	OPTATIVA (6 ECTS)	OPTATIVA (6 ECTS)
MATEMÁTICAS:: FUNDAMENTOS MATEMÁTICOS PARA LA INFORMÁTICA (CFB; 6 ECTS)	MATEMÁTICAS:: ANÁLISIS MATEMÁTICO (CFB; 6 ECTS)	MATEMÁTICAS:: ESTADÍSTICA (CFB; 6 ECTS)	BASES DE DATOS I (OB; 6 ECTS)	BASES DE DATOS II (OB; 6 ECTS)	SISTEMAS INTELIGENTES (OB; 6 ECTS)	OPTATIVA (6 ECTS)	OPTATIVA (6 ECTS)
MATEMÁTICAS:: ÁLGEBRA LINEAL (CFB; 6 ECTS)	INFORMÁTICA: ALGORITMOS Y ESTRUCTURAS DE DATOS I (CFB; 6 ECTS)	ALGORITMOS Y ESTRUCTURAS DE DATOS II (OB; 6 ECTS)	REDES DE COMPUTADORAS I (OB; 6 ECTS)	REDES DE COMPUTADORAS II (OB; 6 ECTS)	CONCURRENCIA Y DISTRIBUCIÓN (OB; 6 ECTS)	SEGURIDAD EN SISTEMAS INFORMÁTICOS (OB; 6 ECTS)	TÉCNICAS DE COMUNICACIÓN Y LIDERAZGO (OB; 6 ECTS)
INFORMÁTICA:: PROGRAMACIÓN I (CFB; 6 ECTS)	PROGRAMACIÓN II (OB; 6 ECTS)	SISTEMAS OPERATIVOS I (OB; 6 ECTS)	SISTEMAS OPERATIVOS II (OB; 6 ECTS)	LENGUAJES DE PROGRAMACIÓN (OB; 6 ECTS)	PROCESADORES DE LENGUAJE (OB; 6 ECTS)	APRENDIZAJE BASADO EN PROYECTOS (OB; 6 ECTS)	**TRABAJO FIN DE GRADO (OB; 12 ECTS)**
FÍSICA:: SISTEMAS DIGITALES (CFB; 6 ECTS)	INFORMÁTICA:: ARQUITECTURA DE COMPUTADORAS I (CFB; 6 ECTS)	ARQUITECTURA DE COMPUTADORAS II (OB; 6 ECTS)	ARQUITECTURAS PARALELAS (OB; 6 ECTS)	HARDWARE DE APLICACIÓN ESPECÍFICA (OB; 6 ECTS)	CENTROS DE DATOS (OB; 6 ECTS)	OPTATIVA (6 ECTS)	
30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS

horas en aula

- 11.01. actividad introductoria (1 hora T + 0.5 horas P)
- 18.01., 25.01., 01.02., 08.02., 15.02., 22.02., 08.03., 15.03., 22.03., 05.04., 12.04., 19.04., 26.04., 03.05., sesiones magistrales + pruebas de respuesta corta
  (01.03. hay las clases del lunes de la misma semana)
  $(14 \cdot 1{,}5 = 21 = 19{,}5 + 1{,}5$ horas)
- 10.05. prueba final (2 horas)
- lunes: 14.01., 21.01., 31.01.(jueves), 04.02., 18.02., 01.03.(viernes), 04.03., 11.03., 18.03., 01.04., 08.04., 15.04., 22.04., 29.04., 06.05., prácticas $(15 \cdot 2 = 30$ horas)
- jueves: 17.01., 24.01., 07.02., 14.02., 21.02., 28.02., 07.03., 14.03., 21.03., 04.04., 11.04., 18.04., 25.04., 02.05., prácticas $(14 \cdot 2 = 28$ horas)

horas de trabajo (este curso)

Actividades introductorias	1.5	(T)	–	1.5	(1.5)
Sesión magistral	18.0	(T)	9.0	27.0	(27.0)
Estudios/actividades previos	–		16.0	16.0	(16.0)
Prácticas en aulas de informática	28.0	(P)	28.0	56.0	(53.0)
Resolución de problemas y/o exercicios	1.5	(T)	15.5	17.0	(19.5)
Tutoría en grupo	1.3		1.2	2.5	(2.5)
Pruebas de respuesta corta	1.5	(T)	–	1.5	(1.5)
Pruebas de respuesta larga	2.0		–	2.0	(2.0)
Informes/memorias de prácticas	–		12.0	12.0	(12.0)
Probas prácticas	1.0	(P)	1.0	2.0	(2.7)
Resolución de problemas y/o exercicios	–		12.0	12.0	(12.0)
Otras	0.5		–	0.5	(0.3)
Suma	55.3		94.7	150.0	(150.0)
primera conv.	50.0		78.0	128.0	

horas del profesor (yo, aproximado, optimista)

1635	$= 218 \cdot 7{,}5$ horas anuales
818	$/2$ docencia
76.5	$22{,}5/240$ segundo cuatrimestre
54	$-22{,}5$ horas presenciales
47	-7 horas preparación clases
32	$-(46/4) - 3$ horas corrección exámenes
2.6	$/46/16$ minutos medio por semana por estudiante

evaluación continua

- (P1) preguntas cortas
- (P2) preguntas largas (hay que aprobar ≥ 4)
- (P3) informes
- (P4) programación (hay que aprobar ≥ 4)
- (P5) análisis
- (P6) presentaciones
- $\min(10, \min(5, 0.2P_1 + 0.4P_2) + \min(4, 0.25P_3 + 0.25P_4) + 0.075P_5 + 0.075P_6) \geq 5$

evaluación no-asistentes

- examen (10.05.) de 3 horas (entre 10:00-14:00) que cubre todo el contenido de la asignatura
- alumnos del curso puente tendrán ciertas consideraciones especiales (quedan por determinar)
- un alumno o bien se autodeclara no-asistente o lo muestra por no asistir a por lo menos 80 % de las actividades presenciales (como mucho se puede faltar a 10 horas de las 49.5 horas de presencialidad principal)

contenido (optimista)

Tema	Contenido
Sistemas concurrentes y distribuidos	Concepto de la programación concurrente y distribuida, Introducción al modelado de sistemas concurrentes y distribuidos, Arquitecturas hardware para la concurrencia y distribución, Herramientas para del desarrollo de aplicaciones concurrentes y distribuidos
Procesos	Concepto de procesos, Planificador, Atomicitad y exclusión mutua, Concurrencia transactional, Reloj y estado distribuido

contenido (optimista)

Tema	Contenido
Sincronización y comunicación	Sincronización y comunicación en sistemas concurrentes y distribuidos, Sincronización y comunicación a nivel bajo y alto, Seguridad y vivacidad en sistemas concurrentes y distribuidos
Herramientas de programación y desarrollo de aplicaciones	Programación concurrente y distribuida con JAVA (y C/C++), Patrones de diseño para el desarrollo de aplicaciones concurrentes y distribuidos, Herramientas y metodologías de diseño, verificación y depuración de aplicaciones concurrentes y distribuidos

Nota

Prácticamente todas las asignaturas optativas en uno u otro aspecto requieren del concepto de concurrencia y distribución en sistemas modernos para lograr sus objetivos específicos.

documento de transparencias

- Este documento crecerá durante el curso,
  ojo, no necesariamente solamente al final.
- Habrá más documentos (capítulos de libros, manuales, etc.)
  con que trabajar durante el curso.
- Los ejemplos de programas y algoritmos serán en inglés.
- Las transparencias no están (posiblemente) ni correctos ni completos.

primera encuesta (curso 2011/12)

- satisfacción global: 7.1
- satisfacción teoría: 6.4
- satisfacción prácticas: 7.4
- tiempo dedicación total: 250 min
- tiempo dedicación teoría: 60 min
- tiempo dedicación prácticas: 200 min
- tiempo dedicación en grupo: 180 min

Los tiempos son valores medianos de la dedicación no-presencial.

reflexión (curso 2011/12)

- Menos carga en prácticas.
- Más ejemplos en teoría.
- Colgar antes las dispositivas (no va a ser).
- Colgar las dispositivas en tochos.
- Hay 4 estudiantes con tiempos muy elevados.

primera encuesta (curso 2012/13)

- satisfacción global: 7.1
- satisfacción teoría: 6.0
- satisfacción prácticas: 7.0
- tiempo dedicación total: 200 min
- tiempo dedicación teoría: 90 min
- tiempo dedicación prácticas: 155 min
- tiempo dedicación en grupo: 35 min

Los tiempos son valores medios de la dedicación no-presencial.

reflexión (curso 2012/13)

- la media de tiempo empleado está bien
- hay gente que no trabaja en grupo fuera del aula
- los viernes son días totalmente normales :–)

introducción

Existen definiciones diversas de los términos

- 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 programación 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 ser ejecutado en hardware paralelo
- un programa distribuido es un programa paralelo diseñado de ser 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?

- selección del algoritmo

problemas

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- división del trabajo

problemas

¿Con qué problemas nos enfrentamos?

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

problemas

¿Con qué problemas nos enfrentamos?

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

problemas

¿Con qué problemas nos enfrentamos?

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

problemas

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- medición de características

problemas

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- medición de características
- depuración del programa

problemas

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- medición de características
- depuración del programa
- (fiabilidad de los componentes)

problemas

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- medición de características
- depuración del programa
- (fiabilidad de los componentes)
- (fiabilidad de la comunicación)

problemas

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados
- medición de características
- depuración del programa
- (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

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 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");
  }
}
```

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`).

¿Qué se comenta?

Existen varias posibilidades de escribir comentarios:

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

- Se usa doxygen o javadoc para generar automáticamente la documentación. Ambos tienen unos comandos para aumentar la documentación.
- Se documenta sobre todo lo que no es obvio y las interfaces
- es decir: respuestas a preguntas del *¿Cómo?* y del *¿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 (como en C++) y el programa principal no es nada más que un método de una clase.

inicialización

- Los objetos en Java siempre tienen valores conocidos, los objetos, es decir, sus miembros 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.
- Variables locales hay que inicializar antes de usarlas, el código se ejecuta cuando la ejecución llega a este punto.

Java, C++, C#

- Java y C++ (o C#) son hasta cierto punto bastante parecidos. (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++.

tipos

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

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 en Java (pero si en C++).
- Los tipos simples no son clases, pero existen para todos los tipos simples clases que implementan el comportamiento de ellos.
- Desde 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 clase, 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
- `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

modificadores de miembros II

- `transient`: excluye un miembro del proceso de conversión en un flujo de bytes si el objeto se salva al disco o se transmite por una conexión (no hay en C++)
- `volatile`: ordena a la máquina virtual de Java que no use ningún tipo de cache para el miembro, así es más probable (aunque no garantizado) que varios hilos vean el mismo valor de una variable; declarando variables del tipo `long` o `double` como `volatile` aseguramos que las operaciones básicas sean atómicas (este tema veremos más adelante más en detalle)

modificadores de 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++)
- `synchronized`: el método pertenece a una región crítica del objeto (no hay en C++)

modificadores de métodos II

- `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` desde 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++.
- desde 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

- Desde 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 diseńar 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 corregir 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;`

agrupación de objetos III

- Java viene con una amplia gama de clases y paquetes predefinidos.
- Se accede a los paquetes con `import`.
- Se accede a los componentes de los paquetes con clasificadores, p.ej., `System.out.println(...)` (desde 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.

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 una estructura de bloques

paquete especial para la programación concurrente

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

  `java.util.concurrent`

- Hay que leer todo su manual.

resumen respecto a concurrencia

- (transient)
- volatile
- synchronized
- try-catch-finally
- finalize
- Thread, Runnable
- java.util.concurrent
- java.util.concurrent.atomic

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 sola intercalación de instrucciones que nos lleva en un fallo.

funcionamiento correcto VII

- Para comprobar si un programa concurrente es incorrecto basta con encontrar una sola 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.

¿Y?

¿El algoritmo de multiplicación con dos hilos de arriba, es correcto parcial? es correcto total?

un programa concurrente

- asumimos que tengamos un programa concurrente que quiere realizar acciones con recursos:
- si los recursos de los diferentes procesos son diferentes no hay problema,
- si dos (o más procesos) quieren manipular el mismo recurso ¿Qué hacemos?

¿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
...\\                           ...\\
```

instrucciones básicas

- obviamente tenemos que asumir que ciertas acciones de un proceso se puede realizar correctamente independientemente de las acciones de los demás procesos
- dichas acciones se llaman “atómicas” (porque son indivisibles) y se garantizan por hardware
- asumimos que podemos acceder a variables de cierto tipo (p.ej. entero) de forma atómica con lectura y escritura (`load` y `store`)

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.
- Vimos y veremos que se pueden realizar protocolos seguros solamente con las instrucciones `load` y `store` de un procesador.
- Las soluciones no suelen ser muy eficientes, especialmente si muchos procesos compiten por la sección crítica. Pero: su desarrollo y la presentación de la solución ayuda en entender el problem principal.
- A veces no hay otra opción disponible.
- 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 true
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.
Si se sabe el número máximo de participantes basta con un número fijo de tickets.

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 (que garantiza una espera mínima para n procesos).

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?
- También existe en la versión `fetch-and-add` que en vez de incrementar suma un valor dado de forma atómica.

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 CAS se pueden realizar los protocolos de entrada y salida. ¿Cómo?

double CAS

Existen también unas mejoras del `CAS`, llamado *double-compare-and-swap* DCAS (Motorola), que realiza dos CAS normales a la vez, o *double-wide compare-and-swap* (Intel/AMD x86), que opera con dos punteros a la vez para el intercambio, o *single-compare double-swap* (Intel itanium), que compara un valor (puntero) pero excribe dos punteros en memoria adyacente.
El código, expresado a alto nivel, para DCAS 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
```

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 I

- 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.

observaciones II

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

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 avance 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.
- Sigue presente el problema de llamar antes a `notify()`, o `notifyAll()` que a `wait()` (*race condition*).

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 estén 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.

Bloqueo en Java

Un ejemplo de un bloqueo en Java muestra el siguiente trozo de código, incluso si se asume que un hilo ya está durmiendo. ¿Por qué?

```
hilo0:

synchronized(A) {
  ...
  synchronized(B) {
    ...
    A.notify();
    B.wait();
  }
}
```

```
hilo1:

synchronized(B) {
  ...
  synchronized(A) {
    ...
    B.notify();
    A.wait();
  }
}
```

seguridad y vivacidad/viveza

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

seguridad: Esa propiedad 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 propiedad 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 inanición 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 espera 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 asignan 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.
- ¡Ojo! ¿Qué se hace si todos tienen la prioridad más alta?

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 concreta:

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 estructura de datos compartida a la cual ambos tienen acceso,
4. si se usa una cola se garantiza un orden temporal
5. ningún dato no está consumido una vez haber sido producido (por lo menos se descarta...)

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

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

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 (terminados) tiene que ser más grande que el número de `signal()`s:

```
        #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 libres 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 señala 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()
```

otras estructuras compartidas

- En un sistema con multiples productores y/o consumidores, puede ser difícil establecer un orden temporal con una semántica adecuada.
- Se puede aflojar la condición de usar una cola, y usar estructuras de datos que permitan más concurrencia.
- Un ejemplo simple serían vectores de contenedores inspeccionados en orden cíclico por los productores y consumidores.

granularidad

Se suele distinguir concurrencia

- de grano fino
  es decir, se aprovecha de la ejecución de operaciones concurrentes a nivel del procesador (hardware)
- a 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, SSE2, SSE3, SSSE3 etc. que realizen según la clasificación SIMD operaciones en varios registros en paralelo.
- Ya están en el marcado los procesadores con multiples núcleos, es decir, se puede programar con varios procesadores que a su vez puedan ejecutar varios 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 suelen 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 ofrecer en vez de aplicaciones para instalar en el cliente, una interfaz hacia un servicio (posiblemente incorporando una red privada virtual) que se ejecute en una conjunto de servidores.
- 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 el mecanismo de conmutación 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/equilibrio entre usuarios/procesos (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 (académicos) que fueron 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 encuentran hoy día en los microprocesadores modernos, sobre todo en los protocolos que controlan la coherencia de los cachés.

memoria compartida homogéneo

- La reordenación automática de instrucciones,
- la división de instrucciones en ensemblador en varios fases de ejecución,
- la intercalación de fases de instrucciones en los procesadores
- el procesamiento en pipelines,
- la aparencia de interrupciones y excepciones,
- la jerarquia de cachés

son propiedades desafiantes para la programación concurrente correcta con procesadores modernos y la traducción de conceptos de alto nivel del lenguaje de programación a código ejecutable no es nada fácil (y no igual en todos los entornos).

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.

algunas herramientas para C/C++ (no exhaustivas)

- ACE (Adaptive Communications Environment)
  `http://www.cs.wustl.edu/~schmidt/ACE.html`
  (usa patrones de diseño de alto nivel, p.ej.; proactor)
- Intel Concurrent Collections (C++, 2012, versión 0.7)
  `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/` (lenguaje de preprocesado para expresar concurrencia)
- Cilk/Cilk++/Cilk Plus (2011)
  `http://software.intel.com/en-us/articles/intel-cilk-plus/`
  (extensión de C/C++ para expresar concurrencia)
- Intel Thread Building Blocks (2012, version 4.0)
  `http://threadingbuildingblocks.org/`
  (C++ template librería para expresar concurrencia)

algunas herramientas para C/C++ (no exhaustivas)

- OpenMP
  `http://openmp.org/wp/`
  (C-preprocessor (+librería embebido) para expresar concurrencia)
- Noble (`www.non-blocking.com`)
- Qt threading library (`http://doc.qt.nokia.com/`)
- pthreads, boost::threads, Zthreads (uso directo de programación multi-hilo)
- próximo/ya estándar de C++ (C++11, 2011)
  (`http://www.open-std.org/jtc1/sc22/wg21/`)

Usa la Red para buscar más información, aquí un ejemplo.

Trazado de rayos concurrente

Rayon con OpenMP

Simulación de partículas concurrente

Pemd con Intel Thread Building Blocks.

modelo de memoria

- Una escritura de una variable `volatile` en Java garantiza que todas las escrituras a variables que aparecen antes en el código se hayan ejecutado ya.
- Entonces otro hilo ve un estado bien definido del hilo escribiendo una variable volatile.
- Es decir, se puede hablar de una relación *ha-pasado-antes* respecto a las escrituras y lecturas.
- Java implementa este modelo, pero es bastante restrictivo respecto a posibles optimizaciones automáticas del compilador.
- C++11 implementa un modelo, que es más flexible para el programador, una primera versión está disponible en g++.4.7 `http://gcc.gnu.org/wiki/Atomic/GCCMM`

Memoria Transaccional

```
P0:
atomic transaction { if( a>b ) c = a-b; }

P1:
atomic transaction { if( a>b ) b++; }
```

Está garantizado que `c` nunca es negativa.

Memoria Transaccional con C++

- la última versión del compilador C++ de GNU (g++-4.7, 2012) da soporte experimental de memoria transaccional.
- `http://gcc.gnu.org/wiki/TransactionalMemory`

comentarios

- Un ingeniero informático no solo usa herramientas ya existentes, sino adapta aquellas que hay a las necesidades concretas y/o inventa nuevas herramientas para avanzar. Eso se llama innovación tecnológica.
- En el ámbito de la concurrencia son más importantes los conceptos que las tecnologías dado que las últimas están actualmente en un proceso de cambio permanente.
- Se debe comparar un programa concurrente/paralelo con el mejor programa secuencial (conocido) para evaluar el rendimiento.

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 asentimiento

- 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*).
- Ejemplos: Reed/Solomon códigos (clásicos), Tornado códigos (finales de los 90), Raptor códigos (*rap*id *tom*ado, en los últimos años))
- Ideas básicas presentadas en pizarra.
- Raptor código: se manda 1.002 MByte, y de cualquier 1 MByte recibido se puede recuperar el mensaje original (con tiempo de cálculo lineal en el longitud del mensaje)
- base para varios nuevos estandares de transmisión de datos en la red

problemática 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 bloqueados 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`).

sincronización del tiempo

problemática

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

- 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.

- Se sincronizan los procesos con los sellos de tiempo que se transmite junto con todos los mensajes.
- Se observa: dos eventos en el mismo procesador 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.

- 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:

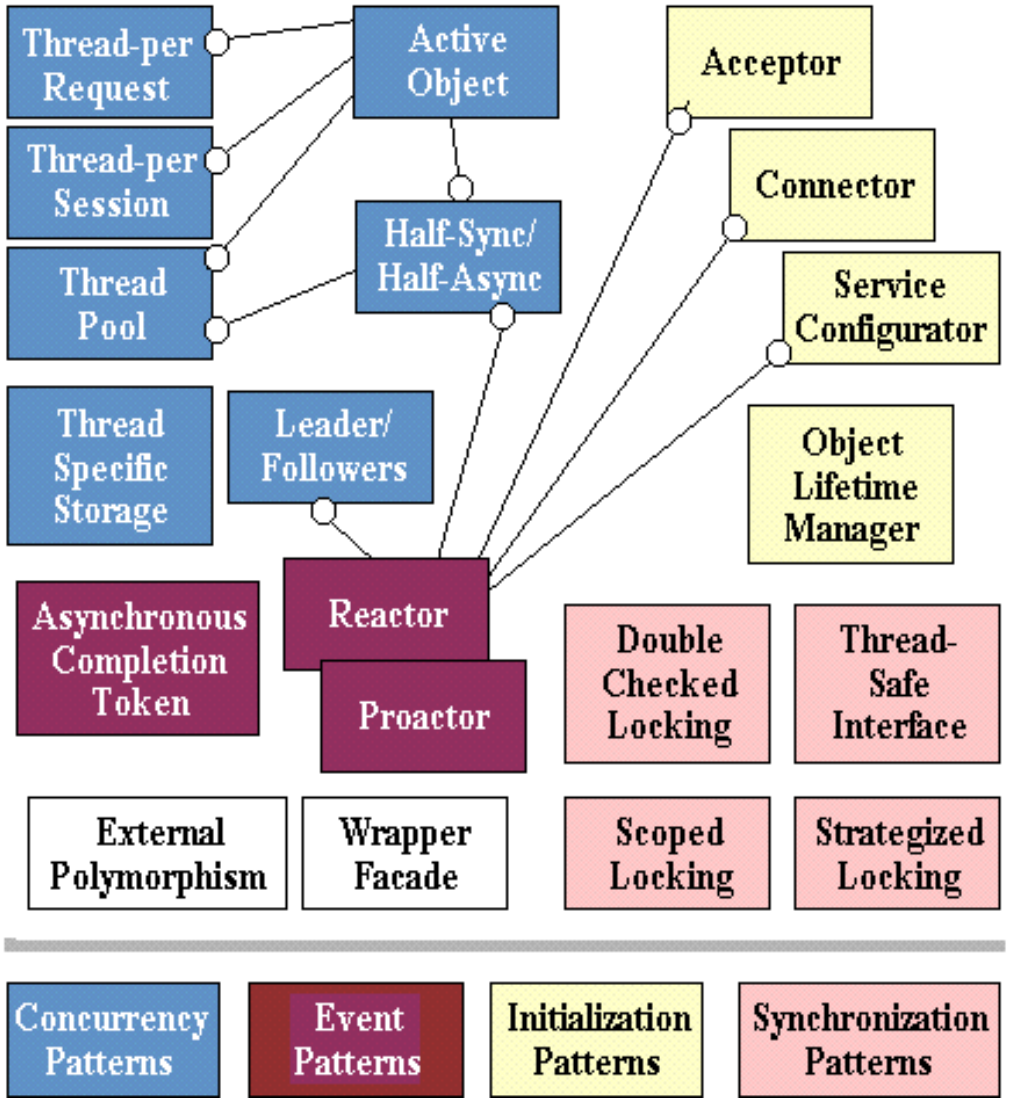
- comprobación directa
- inducción
- contradicción
- contra-ejemplo
- comprobación indirecta
- diagonalización
- reducción
- y más

Algoritmia

paradigmas de desarrollo de algoritmos:

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

Patrones de diseño



http://www.cs.wustl.edu/~schmidt/ACE-papers.html

Patrones

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

- Reactor
- Proactor
- Ficha de terminación asíncrona
- Guardián
- Aviso de hecho (y su uso para el Singleton)

Crítica

- Patrones de diseño no son *el-no-va-más*.
- Muchas veces solamente expresan propiedades que deben estar ya incorporado en el propio lenguaje a alto nivel.
- Sirven como *lenguaje* de comunicación entre programadores, pero hay que tener cuidado que se habla con la misma semántica.
- A veces están demasiado ligados a una implementación en concreta y su uso empuja decisiones a nivel de implementación al nivel de diseño o incluso analysis (fallo típico de un *mero* programador).

Reactor

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*

Reactor

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.

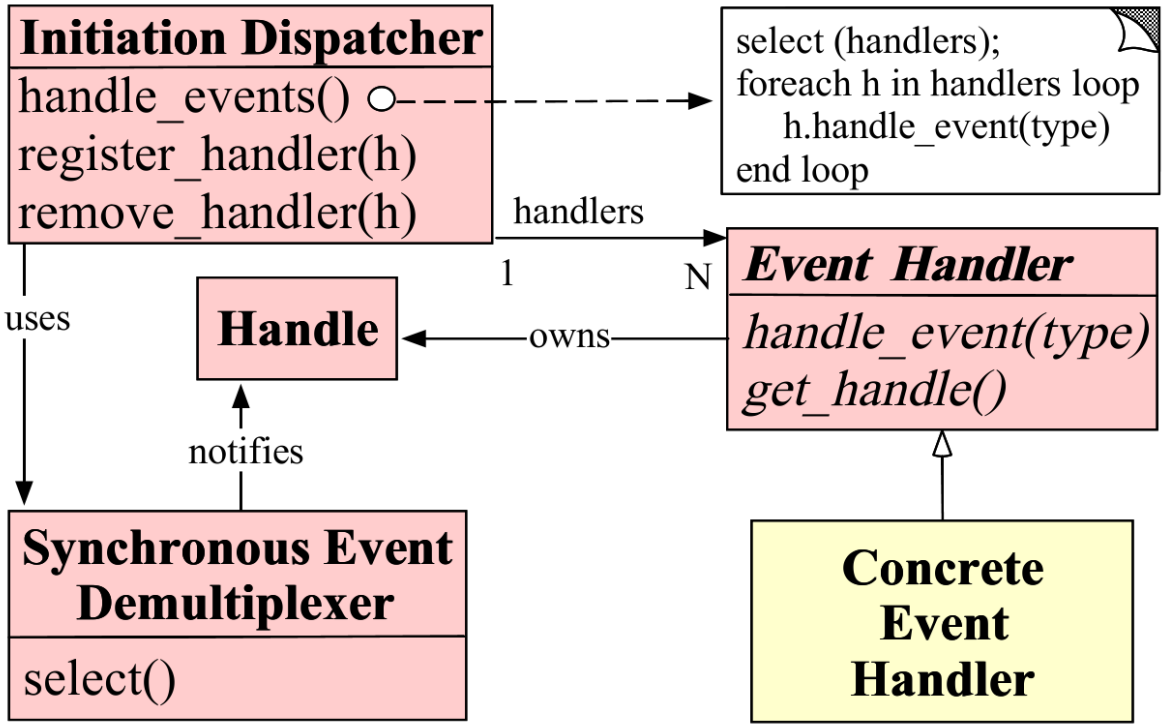
Reactor

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.

Reactor

posible diagrama



(image taken from: D.C. Schmidt, Reactor, 1995)

Reactor

Detalles de la implementación

- Bajo Unix y (parcialmente) bajo Windows se puede aprovechar de la función `select()` para el bucle central.
- Hay que tener cuidado que los eventos en espera tengan posibilidad de llegar al actor por lo menos con espera finita garantizada.
- 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 programación multi-hilo con sincronización).

Proactor

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*

Proactor

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.

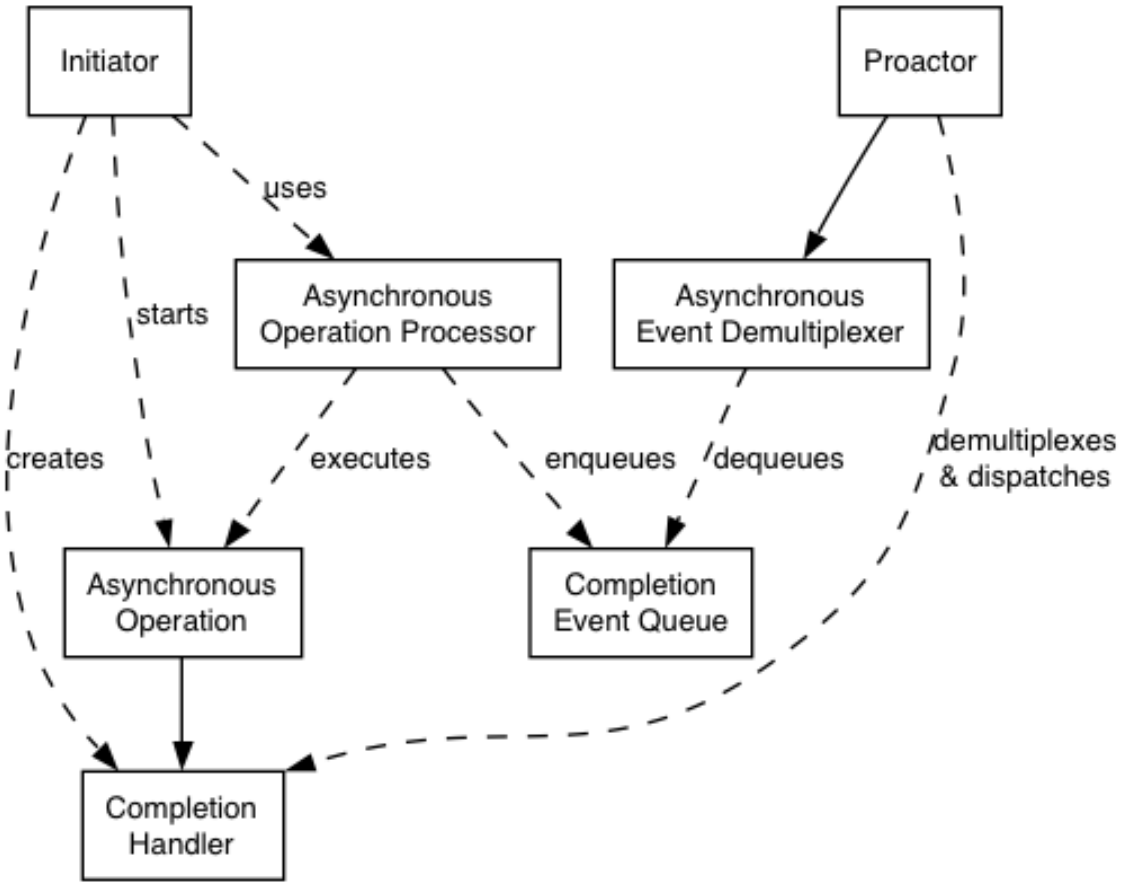
Proactor

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.

Proactor

posible diagrama



(image taken from: Boost library, 2012)

Proactor

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.

Ficha de terminación asíncrono

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*

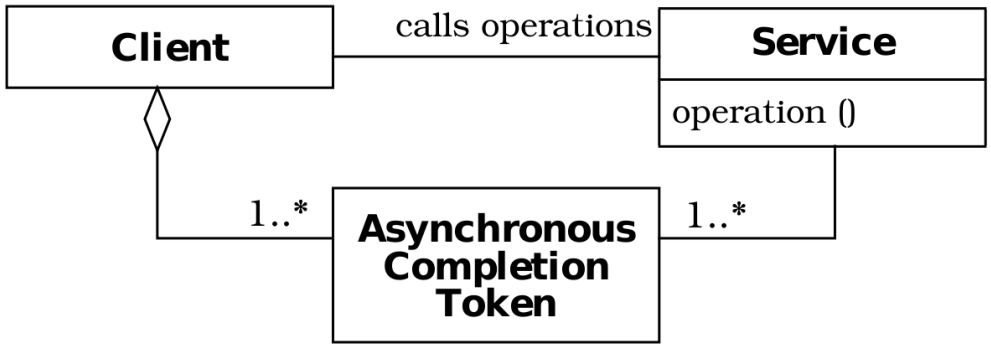
Ficha de terminación asíncrono

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.

Ficha de terminación asíncrono

Posible solución



`http://www.cs.wustl.edu/~schmidt/ACE-papers.html`

- 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.

Ficha de terminación asíncrono

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.

Guardian

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

Guardian

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.

Guardian

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.

Guardian

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.

Guardian

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*).

Aviso de hecho

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

Aviso de hecho

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.

Aviso de hecho

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.

Aviso de hecho

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.

Hay que estar alerta a problemas y cosas nuevas...

- ... mira el artículo de Meyers...
- Scott Meyers and Andrei Alexandrescu. C++ and The Perils of Double-Checked Locking. Dr. Dobb's, The World of Software Development. July 01, 2004
  (versión en PDF en página del curso)
- que demuestra los problemas del patrón en C++03

... que ofrecen soluciones

Pero ahora hay C++11, y con eso funciona lo siguiente, dado que la construcción de objetos estáticos está garantizado de ser seguro con hilos:

```
class Foo
{
public:
    static Foo& instance( void )
    {
        static Foo s_instance;
        return s_instance;
    }
};
```

Más patrones para la concurrencia

- Aceptor–Conector
- Objetos activos
- Monitor
- Mitad-síncrono, mitad-asíncrono
- Líder–y–Seguidores
- Interfaz segura para multi-hilos

Aceptor–Conector

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)

Aceptor–Conector

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.

Aceptor–Conector

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.

Aceptor–Conector

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.

- 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.

Objeto activo

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

Objeto activo

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.

Objeto activo

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.

Objeto activo

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.

Monitor

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

Monitor

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.

Monitor

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).

Monitor

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).

Mitad-síncrono/mitad-asíncrono

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

Mitad-síncrono/mitad-asíncrono

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)

Mitad-síncrono/mitad-asíncrono

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

Mitad-síncrono/mitad-asíncrono

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

Líderes-y-Seguidores

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

Líderes-y-Seguidores

Comportamiento exigido

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

Líderes-y-Seguidores

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

- 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

Interfaz segura para multi-hilos

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

Interfaz segura para multi-hilos

Comportamiento exigido

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

Interfaz segura para multi-hilos

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.

Interfaz segura para multi-hilos

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.