acadèmica

Concurrencia y sistemas distribuidos

Francisco Daniel Muñoz Escoí | Estefanía Argente Villaplana
Agustín Rafael Espinosa Minguet | Pablo Galdámez Saiz
Ana García-Fornes | Rubén de Juan Marín | Juan Salvador Sendra Roig

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CONCURRENCIA Y SISTEMAS DISTRIBUIDOS

Francisco Daniel Muñoz Escoí Estefanía Argente Villaplana
Agustín Rafael Espinosa Minguet Pablo Galdámez Saiz
Ana García-Fornes Rubén de Juan Marín
Juan Salvador Sendra Roig

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Colección Académica

Para referenciar esta publicación, utilice la siguiente cita: MUÑOZ ESCOÍ, F.D. [et al] (2012). *Concurrencia y sistemas distribuidos*. Valencia: Universitat Politècnica

Primera edición 2012 (versión impresa)
Primera edición 2013 (versión electrónica)

Distribución: pedidos@editorial.upv.es
Tel. 96 387 70 12 / www.editorial.upv.es / Ref. editorial: 6084

ISBN: 978-84-8363-986-3 (versión impresa)
ISBN: 978-84-9048-002-1 (versión electrónica)

Prólogo

Este libro va dirigido a los estudiantes de la asignatura de *Concurrencia y Sistemas Distribuidos* de la titulación del Grado en Ingeniería Informática de la Universitat Politècnica de València (UPV), aunque se ha estructurado de forma que cualquier persona interesada en aspectos de programación concurrente y sistemas distribuidos pueda obtener provecho de su lectura.

Los autores de este libro somos profesores de la citada asignatura y, a la hora de confeccionar una relación bibliográfica útil para nuestro alumnado, nos dimos cuenta que en el mercado existe un gran número de libros que abordan aspectos tales como concurrencia, sincronización, sistemas distribuidos, sistemas de tiempo real, administración de sistemas, pero no encontramos ningún libro que tratase todos esos contenidos en conjunto. Este libro pretende, por tanto, dar una visión integradora de las aplicaciones concurrentes y los sistemas distribuidos, ofreciendo también una primera aproximación a las tareas de administración de sistemas (necesarias en sistemas distribuidos).

Conviene destacar que se ha empleado el lenguaje Java para detallar ejemplos de programas concurrentes. La elección de dicho lenguaje de programación se ha debido a las ventajas que aporta para la implementación de programas concurrentes y a su elevado grado de utilización, tanto a nivel profesional como académico. Por ello, se ha incluido un capítulo específico sobre las utilidades que ofrece Java para la implementación de programas concurrentes.

Grado en Ingeniería Informática

El título de Graduado en Ingeniería Informática por la UPV sustituye a los títulos de Ingeniero en Informática, Ingeniero Técnico en Informática de Gestión e Ingeniero Técnico en Informática de Sistemas, como resultado de la adaptación de dichas titulaciones al marco del Espacio Europeo de Educación Superior. Desde enero de 2010 cuenta con la evaluación favorable por parte de la ANECA para su implantación, iniciada en septiembre de 2010.

El Plan de Estudios del título de Grado en Ingeniería Informática de la UPV está organizado en cuatro cursos de 60 ECTS cada uno. Cada ECTS supone 10 horas de docencia presencial y entre 15 y 20 horas para el resto del trabajo del alumno, incluida la evaluación. El Plan de Estudios se estructura en módulos y materias. Cada materia se descompone, a su vez, en una o más asignaturas con tamaños de 4.5, 6 ó 9 ECTS.

Materia de Sistemas Operativos

Dentro del Plan de Estudios del Grado de Ingeniería Informática de la UPV, la asignatura *Concurrencia y Sistemas Distribuidos* pertenece al módulo de Materias Obligatorias y, dentro de dicho módulo, a la materia de *Sistemas Operativos*. Dicha materia comprende el estudio de las características, funcionalidades y estructura de los sistemas operativos así como el estudio de los sistemas distribuidos.

El sistema operativo es un programa que actúa de interfaz entre los usuarios y el hardware del computador, abstrayendo los componentes del sistema informático. De este modo oculta la complejidad del sistema a los usuarios y aplicaciones, y se encarga de la gestión de los recursos, maximizando el rendimiento e incrementado su productividad. Por tanto, el Sistema Operativo se encarga de la gestión de los recursos hardware (CPU, memoria, almacenamiento secundario, dispositivos de entrada/salida, sistema de ficheros), y de controlar la ejecución de los procesos de usuario que necesitan acceder a los recursos hardware.

Por su parte, un sistema distribuido es una colección de ordenadores independientes que ofrecen a sus usuarios la imagen de un sistema coherente único. De forma análoga a como actúa un sistema operativo en una máquina, los sistemas distribuidos se deben encargar de ocultar al usuario las diferencias entre las máquinas y la complejidad de los mecanismos de comunicación. Los sistemas distribuidos deben facilitar el acceso de los usuarios a los recursos remotos, dar trasparencia de distribución (ocultando el hecho de que los procesos y recursos están físicamente distribuidos en diferentes ordenadores), ofrecer estándares para facilitar la interoperabilidad y portabilidad de las aplicaciones, así como ofrecer escalabilidad.

En el Plan de Estudios del Grado de Ingeniería Informática de la UPV, la materia *Sistemas Operativos* es de carácter obligatorio, con una asignación de 12 créditos ECTS y se imparte en segundo curso. Las asignaturas que componen esta materia son dos: Fundamentos de Sistemas Operativos, de 6 ECTS, que se imparte en el primer cuatrimestre de segundo curso; y Concurrencia y Sistemas Distribuidos, de 6 ECTS, que se imparte en el segundo cuatrimestre de ese mismo curso.

En la asignatura de *Fundamentos de Sistemas Operativos* (FSO) se introduce el concepto de sistema operativo, su evolución histórica, así como el funcionamiento y los servicios que proporcionan, detallando la gestión de los procesos, la gestión

de la memoria y la gestión de entrada/salida y ficheros. También se introducen los conceptos de sistemas de tiempo real y la programación de sistemas, detallando el concepto de llamada a sistema y la programación básica de sistemas.

Por su parte, la asignatura de *Concurrencia y Sistemas Distribuidos* (CSD) se centra en describir los principios y técnicas básicas de la programación concurrente, así como las características relevantes de los sistemas distribuidos. De este modo, la asignatura de Concurrencia y Sistemas Distribuidos permitirá que el alumno identifique y resuelva los problemas planteados por la ejecución de múltiples actividades concurrentes en una misma aplicación informática. Tales problemas aparecen a la hora de acceder a recursos compartidos y pueden generar resultados o estados inconsistentes en tales recursos. Para evitarlos se deben emplear ciertos mecanismos de sincronización que el alumno aprenderá a utilizar de manera adecuada. Además, en los sistemas de tiempo real aparecen otras restricciones sobre el uso de estos mecanismos que el alumno identificará y gestionará correctamente.

Por otro lado, el alumno conocerá las diferencias que comporta el diseño y desarrollo de una aplicación distribuida, así como los mecanismos necesarios para proporcionar diferentes tipos de transparencia (replicación, concurrencia, fallos, ubicación, migración, etc.) en este tipo de aplicaciones. Utilizará el modelo cliente/servidor para estructurar estas aplicaciones, por las ventajas que ello comporta a la hora de gestionar los recursos. Conocerá las ventajas que aporta el diseño de algoritmos distribuidos descentralizados, tanto por lo que respecta a la gestión de los fallos como a la escalabilidad de las aplicaciones resultantes. Por último, conocerá las limitaciones existentes a la hora de resolver problemas de sincronización en entornos distribuidos, requiriendo el uso de una ordenación lógica de los eventos en la mayoría de los casos.

Además, se iniciará al alumno en las tareas de administración de sistemas, utilizando el Directorio Activo de Windows Server para este fin.

Ampliación de los contenidos de Sistemas Operativos

Los contenidos de la materia *Sistemas Operativos* se amplían en otras asignaturas en 3^{er} y 4^{o} curso: Tecnologías de los Sistemas de Información en la Red (obligatoria), Administración de Sistemas (del Módulo de Tecnologías de la Información), Diseño de Sistemas Operativos, y Diseño y Aplicaciones de los Sistemas Distribuidos (ambas del Módulo de Ingeniería de Computadores).

Tecnologías de los Sistemas de Información en la Red (obligatoria de 3^{er} curso) introduce las tecnologías y estándares existentes para diseñar, desarrollar, desplegar y utilizar aplicaciones distribuidas. Con ello se extenderá la formación recibida en CSD sobre sistemas distribuidos, desarrollando aplicaciones distribuidas de cierta

complejidad, utilizando el modelo cliente/servidor e identificando diferentes aproximaciones para implantar cada uno de los componentes de estas aplicaciones.

Administración de Sistemas (de 3^{er} curso) desarrolla conceptos y habilidades relacionadas con las tareas de administración de sistemas más habituales en las organizaciones actuales. Entre ellas, se incluyen las habilidades de: instalación, configuración y mantenimiento de sistemas operativos; la gestión de servicios (impresión, ficheros, protocolos); el diseño, planificación e implantación de una política de sistemas (versiones, funcionalidades, licencias, etc.); la administración de dominios de sistema (usuarios, grupos, seguridad); el mantenimiento de copias de respaldo y recuperación ante desastres; la automatización (con el uso de scripts, políticas, etc.); y la formación y soporte al usuario.

Diseño de Sistemas Operativos (de 3^{er} curso) introduce los aspectos de implementación y diseño de sistemas operativos. Se trata de una asignatura fundamentalmente práctica, en las que se trabajará con un sistema operativo real, en concreto con el sistema operativo Linux, viendo con detenimiento los mecanismos que ofrece su núcleo. De este modo, se analizarán las llamadas al sistema que ofrece Linux, la gestión de interrupciones hardware y la gestión de procesos. La comprensión del funcionamiento del sistema operativo permitirá al alumno tomar decisiones de diseño (de aplicaciones) que optimicen el uso de los recursos del computador.

Por último, *Diseño y Aplicaciones de los Sistemas Distribuidos* (de 4^{o} curso) profundiza en los aspectos de comunicación en sistemas distribuidos (modelos cliente/servidor, modelos orientados a objetos, modelos de grupos), en los conceptos de seguridad, así como en las tecnologías para la integración de aplicaciones en la web y los aspectos básicos de diseño de aplicaciones distribuidas, tales como replicación y caching.

Conocimientos recomendados

Respecto a los conocimientos recomendados para el seguimiento adecuado de la asignatura de *Concurrencia y Sistemas Distribuidos*, la primera parte de sus unidades didácticas se centra en los problemas de concurrencia y sincronización en un modelo de memoria compartida. Para ello, los alumnos deberán hacer uso de los conceptos de hilo y proceso estudiados en *Fundamentos de Sistemas Operativos*.

Por otro lado, para la realización de las prácticas, así como para el estudio, comprensión y resolución de los problemas y casos de estudio, es recomendable que los alumnos tengan un aceptable nivel de programación utilizando Java. Dicho nivel se debe haber alcanzado en las asignaturas de programación de primer curso (*Introducción a la Informática y a la Programación*; *Programación*; y *Fundamentos de Computadores*) y la correspondiente de primer cuatrimestre de segundo curso (*Lenguajes, Tecnologías y Paradigmas de la Programación*).

Índice general

ÍNDICE GENERAL V

ÍNDICE DE FIGURAS XI

ÍNDICE DE TABLAS XVII

1 PROGRAMACIÓN CONCURRENTE 1

1.1 Introducción 1
1.2 Definición 1
1.3 Aplicaciones concurrentes. 4
1.3.1 Ventajas e inconvenientes 4
1.3.2 Aplicaciones reales 6
1.3.3 Problemas clásicos 9
1.4 Tecnología Java 14
1.4.1 Concurrencia en Java 14
1.4.2 Gestión de hilos de ejecución 15
1.5 Resumen 18

2 COOPERACIÓN ENTRE HILOS 19

2.1 Introducción 19
2.2 Hilos de ejecución 20
2.2.1 Ciclo de vida de los hilos Java 21

2.3 Cooperación entre hilos 24
2.3.1 Comunicación 25
2.3.2 Sincronización 26
2.4 Modelo de ejecución 27
2.5 Determinismo 30
2.6 Sección crítica 34
2.6.1 Gestión mediante *locks* 37
2.7 Sincronización condicional 40
2.8 Resumen 40

3 PRIMITIVAS DE SINCRONIZACIÓN 43

3.1 Introducción 43
3.2 Monitores 46
3.2.1 Motivación 46
3.2.2 Definición 49
3.2.3 Ejemplos 50
3.2.4 Monitores en Java 53
3.3 Variantes. 56
3.3.1 Variante de Brinch Hansen 59
3.3.2 Variante de Hoare 60
3.3.3 Variante de Lampson y Redell 63
3.4 Invocaciones anidadas 67
3.5 Resumen 68

4 INTERBLOQUEOS 71

4.1 Introducción 71
4.2 Definición de interbloqueo 72
4.2.1 Condiciones de Coffman 74
4.2.2 Ejemplos 74
4.3 Representación gráfica 76
4.3.1 Algoritmo de reducción de grafos 78
4.4 Soluciones 81
4.4.1 Prevención 83
4.4.2 Evitación 87

4.4.3 Detección y recuperación 88
4.4.4 Ejemplos de soluciones 89
4.5 Resumen 91

5 BIBLIOTECA java.util.concurrent 93
5.1 Introducción 93
5.2 Inconvenientes de las primitivas básicas de Java 94
5.3 La biblioteca java.util.concurrent 95
5.3.1 Locks 95
5.3.2 Condition y monitores 99
5.3.3 Colecciones concurrentes 101
5.3.4 Variables atómicas 104
5.3.5 Semáforos 109
5.3.6 Barreras 113
5.3.7 Ejecución de hilos 118
5.3.8 Temporización precisa 119
5.4 Resumen 120

6 SISTEMAS DE TIEMPO REAL 123
6.1 Introducción 123
6.2 Análisis básico 125
6.3 Ejemplo de cálculo de los tiempos de respuesta 128
6.4 Compartición de recursos 129
6.4.1 Protocolo de techo de prioridad inmediato 132
6.4.2 Propiedades 134
6.4.3 Cálculo de los factores de bloqueo 135
6.4.4 Cálculo del tiempo de respuesta con factores de bloqueo 137
6.5 Resumen 139

7 SISTEMAS DISTRIBUIDOS 141
7.1 Introducción 141
7.2 Definición de sistema distribuido 142
7.3 Objetivos de los Sistemas Distribuidos 145
7.3.1 Acceso a recursos remotos 145
7.3.2 Transparencia de distribución 146

7.3.3 Sistemas abiertos 152
7.3.4 Sistemas escalables 154
7.4 Dificultades en el desarrollo de Sistemas Distribuidos 163
7.5 Resumen 164

8 COMUNICACIONES 167
8.1 Introducción 167
8.2 Arquitectura en niveles (TCP/IP) 168
8.3 Llamada a procedimiento remoto (RPC) 170
8.3.1 Pasos en una RPC 172
8.3.2 Paso de argumentos 174
8.3.3 Tipos de RPC 175
8.4 Invocación a objeto remoto 177
8.4.1 Visión de usuario 180
8.4.2 Creación y registro de objetos 181
8.4.3 Detalles de la invocación remota. 183
8.4.4 Otros aspectos. 189
8.4.5 RMI (Remote Method Invocation) 190
8.5 Comunicación basada en mensajes 195
8.5.1 Sincronización 195
8.5.2 Persistencia 198
8.6 Resumen 199

9 SINCRONIZACIÓN DISTRIBUIDA 201
9.1 Introducción 201
9.2 Relojes 202
9.2.1 Algoritmos de sincronización 203
9.2.2 Relojes lógicos. 207
9.2.3 Relojes vectoriales 211
9.3 Estado global. 213
9.3.1 Corte o imagen global 214
9.3.2 Algoritmo de Chandy y Lamport 215
9.4 Elección de líder 219
9.4.1 Algoritmo “Bully” 220
9.4.2 Algoritmo para anillos 221

9.5 Exclusión mutua 222
9.5.1 Algoritmo centralizado 222
9.5.2 Algoritmo distribuido 223
9.5.3 Algoritmo para anillos 224
9.5.4 Comparativa 225
9.6 Resumen 226

10 GESTIÓN DE RECURSOS 229
10.1 Introducción 229
10.2 Nombrado 230
10.2.1 Conceptos básicos 230
10.2.2 Espacios de nombres 232
10.3 Servicios de localización 237
10.3.1 Localización por difusiones 238
10.3.2 Punteros adelante 239
10.4 Servicios de nombres jerárquicos: DNS 242
10.5 Servicios basados en atributos: LDAP 245
10.6 Resumen 249

11 ARQUITECTURAS DISTRIBUIDAS 251
11.1 Introducción 251
11.2 Arquitecturas *software* 252
11.3 Arquitecturas de sistema 255
11.3.1 Arquitecturas centralizadas 257
11.3.2 Arquitecturas descentralizadas 261
11.4 Resumen 272

12 DIRECTORIO ACTIVO 275
12.1 Introducción 275
12.2 Servicios de dominio 275
12.2.1 Controlador de dominio 276
12.2.2 Árboles y bosques 277
12.2.3 Principales protocolos empleados 279

12.3 Servicios de directorio 279
12.3.1 Arquitectura y componentes 280
12.3.2 Objetos del directorio 281
12.4 Gestión de permisos 284
12.4.1 Permisos para archivos 285
12.4.2 Permisos para carpetas 286
12.4.3 Listas de control de acceso 288
12.4.4 Uso de las ACL 290
12.4.5 Diseño de ACL 293
12.5 Recursos compartidos 294
12.5.1 Creación del recurso compartido 295
12.5.2 Asignación de un identificador de unidad 295
12.5.3 Autenticación y autorización 296
12.6 Resumen 297

BIBLIOGRAFÍA 301

ÍNDICE ALFABÉTICO 311

Índice de figuras

1.1. Uso de un buffer de capacidad limitada. 10
1.2. Problema de los cinco filósofos. 13
1.3. Ejemplo de creación de hilos. 17

2.1. Diagrama de estados de los hilos en Java. 22
2.2. Clase Node 28
2.3. Clase Queue. 29
2.4. Contador no determinista. 31
2.5. Protocolos de entrada y salida a una sección crítica. 35
2.6. Contador determinista. 39

3.1. Ejemplo de monitor. 51
3.2. Monitor del simulador. 53
3.3. Segundo monitor del simulador. 53
3.4. Monitor Buffer en Java. 56
3.5. Monitor Java para el simulador de la colonia de hormigas. 57
3.6. Ejemplo de monitor con reactivación en cascada. 57
3.7. Ejemplo de monitor con ambos tipos de reactivación. 58
3.8. Monitor que implanta el tipo o clase `Semaphore`. 60

3.9. Monitor `BoundedBuffer` (variante de Brinch Hansen). 61
3.10. Monitor `BoundedBuffer` (variante de Hoare). 62
3.11. Monitor `SynchronousLink` (variante de Hoare). 64
3.12. Monitor `BoundedBuffer` en Java. 66
3.13. Monitor `BoundedBuffer` incorrecto en Java. 67

4.1. Grafo de asignación de recursos. 77
4.2. Algoritmo de reducción de grafos. 78
4.3. Traza del algoritmo de reducción (inicial). 79
4.4. Traza del algoritmo de reducción (iteración 1). 80
4.5. Traza del algoritmo de reducción (iteración 2). 81
4.6. Traza del algoritmo de reducción (iteración 3). 82
4.7. Segunda traza de reducción (estado inicial). 83
4.8. Segunda traza de reducción (iteración 1). 84
4.9. Problema de los cinco filósofos. 90

5.1. Sección crítica protegida por un `ReentrantLock`. 98
5.2. Monitor `BoundedBuffer`. 100
5.3. Gestión de un buffer de capacidad limitada. 104
5.4. Contador concurrente. 108
5.5. Solución con semáforos al problema de productores-consumidores. 111
5.6. Sincronización con `CyclicBarrier` (incorrecta). 114
5.7. Sincronización con `CyclicBarrier` (correcta). 115
5.8. Sincronización con `CountDownLatch`. 117

6.1. Representación gráfica de los atributos de las tareas. 126
6.2. Cronograma de ejecución. 128

6.3. Ejemplo de inversión de prioridades. 130
6.4. Ejemplo de interbloqueo 131
6.5. Protocolo del techo de prioridad inmediato. 133
6.6. Evitación de interbloqueo con el protocolo del techo de prioridad inmediato 134
6.7. Sistema formado por tres tareas y dos semáforos. 137

7.1. Arquitectura de un sistema distribuido. 144
7.2. Transparencia de ubicación en una RPC. 148

8.1. Propagación de mensajes (Arquitectura TCP/IP). 169
8.2. Visión basada en protocolos de la arquitectura TCP/IP. 170
8.3. Ubicación del middleware en una arquitectura de comunicaciones. . 170
8.4. Pasos en una llamada a procedimiento remoto. 173
8.5. RPC convencional. 175
8.6. RPC asincrónica. 176
8.7. RPC asincrónica diferida. 176
8.8. Invocación local y remota. 178
8.9. Funcionamiento básico de una ROI. 180
8.10. Creación de objetos remotos a solicitud del cliente. 181
8.11. Creación de objetos remotos por parte del servidor. 182
8.12. Detalles de una invocación remota. 183
8.13. Referencia directa. 185
8.14. Referencia indirecta. 186
8.15. Referencia a través de un localizador. 186
8.16. Paso de objetos por valor. 187
8.17. Paso por referencia de un objeto local. 188

8.18. Paso por referencia de un objeto remoto. 188
8.19. Ejemplo de paso de argumentos. 189
8.20. Utilización del registro en RMI. 191
8.21. Diagrama de ejecución del ejemplo. 194
8.22. Ejemplo de comunicación asincrónica. 196
8.23. Comunicación sincrónica basada en envío. 196
8.24. Comunicación sincrónica basada en entrega. 197
8.25. Comunicación sincrónica basada en procesamiento. 197

9.1. Sincronización con el algoritmo de Cristian. 205
9.2. Relojes lógicos de Lamport. 209
9.3. Relojes lógicos de Lamport generando un orden total. 210
9.4. Relojes vectoriales. 212
9.5. Corte preciso de una ejecución. 214
9.6. Corte consistente de una ejecución. 214
9.7. Corte inconsistente de una ejecución. 215
9.8. Traza del algoritmo de Chandy y Lamport (Pasos 1 a 3). 217
9.9. Traza del algoritmo de Chandy y Lamport (Pasos 4 a 6). 217
9.10. Traza del algoritmo de Chandy y Lamport (Pasos 7 a 9). 218
9.11. Traza del algoritmo de Chandy y Lamport (Estado global). 219

10.1. Directorios y entidades en un espacio de nombres. 233
10.2. Niveles en un espacio de nombres jerárquico. 234
10.3. Inserción de un puntero adelante. 239
10.4. Recorte de una cadena de punteros adelante. 240
10.5. Ejemplo de fichero de zona DNS. 246

11.1. Arquitectura software en niveles. 252
11.2. Arquitectura software basada en eventos. 255
11.3. Roles cliente y servidor en diferentes invocaciones. 258
11.4. Secuencia petición-respuesta en una interacción cliente-servidor. . . 258
11.5. Cinco ejemplos de despliegue de una arquitectura en capas. 260
11.6. Distribución horizontal en un servidor web replicado. 262
11.7. Grados de centralización en un sistema P2P. 264
11.8. Ubicación de recursos en el sistema Chord. 267

12.1. Elementos en un dominio. 276
12.2. Ejemplo de bosque de dominios. 278
12.3. Arquitectura de los servicios de directorio en un DC. 280
12.4. Relaciones entre los diferentes tipos de permisos (para archivos). . 286
12.5. Relaciones entre los diferentes tipos de permisos (para carpetas). . 287
12.6. Recurso compartido. 294

Índice de tablas

1.1. Variantes de la definición de hilos. 15

3.1. Tabla de métodos. 51
3.2. Tabla de métodos del monitor `Hormigas`. 52
3.3. Características de las variantes de monitor. 59
3.4. Traza con la variante de Hoare. 63

5.1. Métodos de la clase `ReentrantLock`. 97
5.2. Métodos de la interfaz `Condition`. 99
5.3. Métodos de la interfaz `BlockingQueue<E>`. 102
5.4. Clases del *package* `java.util.concurrent.atomic`. 106
5.5. Métodos de la clase `AtomicInteger`. 107

7.1. Tipos de transparencia. 146

9.1. Comparativa de algoritmos de exclusión mutua. 225

11.1. Clases de sistemas P2P. 268

12.1. Ejemplo de ACL con entradas explícitas y heredadas. 289
12.2. Ejemplo de ACL con desactivación de herencia. 289

12.3. Ejemplo de ACL. 291

Unidad 1

PROGRAMACIÓN CONCURRENTE

1.1 Introducción

Esta unidad didáctica presenta el concepto de *programación concurrente*, proporcionando su definición informal en la sección 1.2. La sección 1.3 discute las principales ventajas e inconvenientes que ofrece este tipo de programación cuando es comparada frente a la programación secuencial. Además, se presentan algunos ejemplos de aplicaciones que típicamente serán concurrentes y se discute por qué resulta ventajoso programarlas de esa manera. Por último, la sección 1.4 ofrece una primera introducción a los mecanismos incorporados en el lenguaje *Java* para soportar la programación concurrente.

1.2 Definición

Un *programa secuencial* [Dij71] es aquel en el que sus instrucciones se van ejecutando en serie, una tras otra, desde su inicio hasta el final. Por tanto, en un programa secuencial solo existe una única actividad o hilo de ejecución. Es la forma tradicional de programar, pues los primeros ordenadores solo tenían un procesador y los primeros sistemas operativos no proporcionaron soporte para múltiples actividades dentro de un mismo proceso.

Por el contrario, en un *programa concurrente* [Dij68] se llegarán a crear múltiples actividades que progresarán de manera simultánea. Cada una de esas actividades será secuencial pero ahora el programa no avanzará siguiendo una única secuencia,

pues cada actividad podrá seguir una serie de instrucciones distinta y todas las actividades avanzan de manera simultánea (o, al menos, esa es la imagen que ofrecen al usuario). Para que en un determinado proceso haya múltiples actividades, cada una de esas actividades estará soportada por un *hilo de ejecución*. Para que un conjunto de actividades constituya un programa concurrente, todas ellas deben cooperar entre sí para realizar una tarea común.

Para ilustrar el concepto de programa o proceso concurrente, podríamos citar a un procesador de textos. Existen varios de ellos actualmente (MS Word, LibreOffice Writer, Apple Pages, ...). La mayoría son capaces de realizar múltiples tareas en segundo plano, mientras se va escribiendo (para ello se necesita que haya un hilo de ejecución que se dedique a cada una de esas acciones, mientras otro atienda a la entrada proporcionada por el usuario). Algunos ejemplos de tareas realizadas por estos hilos son: revisión gramatical del contenido del documento, grabación del contenido en disco de manera periódica para evitar las pérdidas de texto en caso de corte involuntario de alimentación, actualización de la ventana en la que se muestra el texto introducido, etc. Como se observa en este ejemplo, en este programa (formado por un solo fichero ejecutable) existen múltiples hilos de ejecución y todos ellos cooperan entre sí y comparten la información manejada (el documento que estemos editando).

Presentemos seguidamente un ejemplo de una ejecución no concurrente. Asumamos que en una sesión de laboratorio abrimos varias consolas de texto en Linux para realizar las acciones que nos pida algún enunciado y probarlas. En este segundo caso, seguiríamos utilizando un único programa (un intérprete de órdenes; p. ej., el `bash`) y tendríamos múltiples actividades (es decir, habría varias actividades ejecutando ese único programa), pero esas actividades no colaborarían entre sí. Cada consola abierta la estaríamos utilizando para lanzar una serie distinta de órdenes y las órdenes lanzadas desde diferentes consolas no interactuarían entre sí. Serían múltiples instancias de un mismo programa en ejecución, generando un conjunto de procesos **independientes**. Se ejecutarían simultáneamente, pero cada uno de ellos no dependería de los demás. Eso no es una aplicación concurrente: **para que haya concurrencia debe haber cooperación entre las actividades**. Algo similar ocurriría si en lugar de lanzar múltiples consolas hubiésemos lanzado múltiples instancias del navegador. Cada ventana abierta la estaríamos utilizando para acceder a una página distinta y no tendría por qué existir ningún tipo de interacción o cooperación entre todos los procesos navegadores generados.

En esta sección se ha asumido hasta el momento que para generar una aplicación informática basta con un único programa. Esto no tiene por qué ser siempre así. De hecho, algunas aplicaciones pueden estructurarse como un conjunto de componentes y cada uno de esos componentes puede estar implantado mediante un programa distinto. En general, cuando hablemos de una *aplicación concurrente* asumiremos que está compuesta por un conjunto de actividades que cooperan entre sí para realizar una tarea común. Estas actividades podrán ser: múltiples hilos

en un mismo proceso, múltiples procesos en una misma máquina o incluso múltiples procesos en máquinas distintas; pero en todos los casos existirá cooperación entre las actividades. Por el contrario, en una *aplicación secuencial* únicamente existirá un solo proceso y en tal proceso solo habrá un único hilo de ejecución.

Para soportar la concurrencia se tendrá que ejecutar ese conjunto de actividades de manera simultánea o paralela. Para ello existen dos mecanismos:

- *Paralelismo real*: Se dará cuando tengamos disponibles múltiples procesadores, de manera que ubicaremos a cada actividad en un procesador diferente. En las arquitecturas modernas, los procesadores pueden tener múltiples núcleos. En ese caso, basta con asignar un núcleo diferente a cada actividad. Todas ellas podrán avanzar simultáneamente sin ningún problema.

  Si solo disponemos de ordenadores con un solo procesador y un único núcleo, podremos todavía lograr la concurrencia real utilizando múltiples ordenadores. En ese caso la aplicación concurrente estará formada por múltiples procesos y estos cooperarán entre sí intercambiando mensajes para comunicarse.

- *Paralelismo lógico*: Como los procesadores actuales son rápidos y las operaciones de E/S suspenden temporalmente el avance del hilo de ejecución que las haya programado, se puede intercalar la ejecución de múltiples actividades y ofrecer la imagen de que éstas progresan simultáneamente. Así, mientras se sirve una operación de E/S, el procesador queda libre y se puede elegir a otra actividad preparada para ejecutarla durante ese intervalo. Este es el principio seguido para proporcionar *multiprogramación* (técnica de multiplexación que permite la ejecución simultánea de múltiples procesos en un único procesador, de manera que parece que todos los procesos se están ejecutando a la vez, aunque hay un único proceso en el procesador en cada instante de tiempo). Por ello, el hecho de que únicamente se disponga de un solo procesador en un determinado ordenador no supone ninguna restricción que impida implantar aplicaciones concurrentes.

  El usuario será incapaz de distinguir entre un sistema informático con un solo procesador y un solo núcleo de otro que tenga múltiples procesadores, pues los sistemas operativos ocultan estos detalles. En cualquiera de los dos casos, el usuario percibe la imagen de que múltiples actividades avanzan simultáneamente.

Ambos tipos de paralelismo pueden combinarse sin ninguna dificultad. Por ello, si en un determinado ordenador tenemos un procesador con dos núcleos, el número de actividades que podremos soportar en una aplicación concurrente que ejecutemos en él no tendrá por qué limitarse a dos. Al combinar el paralelismo real con el lógico podremos utilizar tantos hilos como sea necesario en esa aplicación.

1.3 Aplicaciones concurrentes

Esta sección estudia con mayor detenimiento algunos aspectos complementarios de la programación concurrente, como son sus principales ventajas e inconvenientes, así como otros ejemplos de aplicaciones, tanto reales como académicos. Estos últimos ilustran algunas de las dificultades que encontraremos a la hora de desarrollar una aplicación concurrente.

1.3.1 Ventajas e inconvenientes

La programación concurrente proporciona las siguientes ventajas:

- *Eficiencia*: El hecho de disponer de múltiples actividades dentro de la aplicación permite que ésta pueda concluir su trabajo de manera más rápida. El hecho de que alguna actividad se suspenda al utilizar (p.ej., al acceder al disco para leer parte de un fichero) o al solicitar (p.ej., al utilizar la operación P() sobre un semáforo) algún recurso no suspende a toda la aplicación y con ello esta última podrá seguir avanzando. En una aplicación concurrente resulta sencillo que se utilicen múltiples recursos simultáneamente (ficheros, semáforos, memoria, procesador, red,...).

- *Escalabilidad*: Si una determinada aplicación puede diseñarse e implantarse como un conjunto de componentes que interactúen y colaboren entre sí, generando al menos una actividad por cada componente, se facilitará la distribución de la aplicación sobre diferentes ordenadores. Para ello bastaría con instalar cada componente en un ordenador distinto, generando una *aplicación distribuida*. De esta manera, la cantidad de recursos de cómputo que se podrán utilizar simultáneamente se incrementará. Además, si el número de usuarios de esa aplicación también se incrementara significativamente, se podrían utilizar técnicas de *replicación* [Sch90, BMST92], incrementando su capacidad de servicio. Así mejora la escalabilidad de una aplicación.

  En general, se dice que un *sistema* es *escalable* [Bon00] cuando es capaz de gestionar adecuadamente cargas crecientes de trabajo o cuando tiene la capacidad para crecer y adaptarse a tales cargas. En el caso de las aplicaciones concurrentes ambas interpretaciones se pueden cumplir combinando la distribución y replicación de componentes.

- *Gestión de las comunicaciones*: El uso eficiente de los recursos, ya comentado en la primera ventaja citada en este apartado, permite que aquellos recursos relacionados con la comunicación entre actividades sean explotados de manera sencilla en una aplicación concurrente. Si la comunicación está basada en el intercambio de mensajes a través de la red, esta comunicación no implicará que toda la aplicación se detenga esperando alguna respuesta. Por ello, el uso de mecanismos sincrónicos de comunicación entre actividades

se puede integrar de manera natural en una aplicación concurrente. En la Unidad 8 se detallarán diferentes tipos de mecanismos de comunicación.

- *Flexibilidad*: Las aplicaciones concurrentes suelen utilizar un diseño modular y estructurado, en el que se presta especial atención a qué es lo que debe hacer cada actividad, qué recursos requerirá y qué módulos del programa necesitará ejecutar. Por ello, si los requisitos de la aplicación cambiasen, resultaría relativamente sencillo identificar qué módulos tendrían que ser adaptados para incorporar esas variaciones en la especificación y a qué actividades implicarían. Como resultado de este diseño estructurado y cuidadoso, se generarán aplicaciones flexibles y adaptables.
- *Menor hueco semántico*: Un buen número de aplicaciones informáticas requieren el uso de varias actividades simultáneas (por ejemplo, en un videojuego suele utilizarse un hilo por cada elemento móvil que haya en la pantalla; en una hoja de cálculo tenemos un hilo para gestionar la entrada de datos, otro para recálculo, otro para la gestión de los menús, otro para la actualización de las celdas visibles, ...). Si estas aplicaciones se implantan como programas concurrentes resulta sencillo su diseño. Por el contrario, si se intentara implantarlas como un solo programa secuencial la coordinación entre esas diferentes funcionalidades resultaría difícil.

No obstante, la programación concurrente también presenta algunos inconvenientes que conviene tener en cuenta:

- *Programación delicada*: Durante el desarrollo de aplicaciones concurrentes pueden llegar a surgir algunos problemas. El más importante se conoce como "*condición de carrera*" (a estudiar en la unidad 2) y puede generar inconsistencias imprevistas en el valor de las variables o atributos compartidos entre las actividades, cuando éstas los modifiquen. Un segundo problema son los *interbloqueos* [CES71], que se analizarán en la Unidad 4. Por ello, deben conocerse los problemas potenciales que entraña la programación concurrente y deben tomarse las precauciones oportunas para evitar que aparezcan.
- *Depuración compleja*: Una aplicación concurrente, al estar compuesta por múltiples actividades, puede intercalar de diferentes maneras en cada ejecución las sentencias que ejecuten cada una de sus actividades. Por ello, aunque se proporcionen las mismas entradas a la aplicación, los resultados que ésta genere pueden llegar a ser distintos en diferentes ejecuciones. Si alguno de estos resultados fuese incorrecto, resultaría bastante difícil la reproducción de esa ejecución. Además, como para depurar cualquier aplicación informática se suelen utilizar herramientas especializadas (diferentes tipos de depuradores), la propia gestión del depurador podría evitar que se diera la traza que provocó ese error. Esto ilustra que las aplicaciones concurrentes no tienen una depuración sencilla y que los mecanismos empleados en sus depuradores no siempre serán idénticos a los utilizados sobre aplicaciones secuenciales.

A pesar de estos inconvenientes existe un interés creciente en el desarrollo de aplicaciones concurrentes. Una buena parte de las aplicaciones actuales se estructuran en múltiples componentes que pueden ser desplegados en ordenadores diferentes, es decir, son aplicaciones concurrentes distribuidas cuyos componentes necesitan intercomunicarse a través de la red. También están apareciendo múltiples dispositivos móviles con procesadores aceptables y con discos duros o memorias flash con suficiente capacidad de almacenamiento. Estos elementos ("*tablets*", "*netbooks*", "*ultrabooks*", teléfonos inteligentes,...) interactúan tanto con su entorno como con otros dispositivos móviles y es habitual que utilicen múltiples actividades en sus aplicaciones, tal como sugiere un modelo de programación concurrente. A su vez, no es raro que los procesadores utilizados tanto en los ordenadores personales tradicionales como en ese nuevo conjunto de ordenadores "ligeros" dispongan de múltiples núcleos. Con ello, hoy en día no supone ningún inconveniente la ejecución paralela de múltiples actividades, tanto de manera lógica como real.

1.3.2 Aplicaciones reales

Pasemos a ver seguidamente algunos ejemplos de aplicaciones concurrentes reales. Estos ejemplos ilustran los conceptos presentados hasta el momento en esta unidad, pero también proporcionan la base para entender algunos de los aspectos analizados en las unidades posteriores en las que presentamos los problemas que plantea la programación concurrente así como los mecanismos necesarios para solucionarlos:

1. *Programa de control del acelerador lineal Therac-25* [LT93]. Los aceleradores lineales se utilizan en los tratamientos de radioterapia para algunos tipos de cáncer, eliminando mediante ciertos clases de radiación (radiación de electrones, rayos X y rayos gamma, principalmente) a las células cancerígenas. El Therac-25 fue un acelerador lineal desarrollado entre 1976 y 1982, tomando como base los diseños de otros aceleradores más antiguos (el Therac-6 y el Therac-20). En el Therac-25 se podían utilizar dos tipos de radiación (de electrones y gamma) abarcando un amplio intervalo de intensidades. En los modelos anteriores, el acelerador disponía de sistemas de control por hardware que evitaban cualquier tipo de disfunción (parando el acelerador en caso de error). En el Therac-25 esa gestión se integró en el software.

   El tratamiento se realizaba en una sala aislada. El operador del sistema configuraba adecuadamente el acelerador una vez el paciente estuviese preparado. Para ello se debía especificar el tipo de radiación a utilizar, la potencia a emplear, la duración de la sesión, etc. El sistema tardaba algunos segundos en responder a esa configuración. Una vez transcurría ese tiempo, empezaba el tratamiento. El programa de control se encargaba de monitorizar todos los componentes del sistema: posición y orientación del acelerador, potencia de la radiación, posición de los escudos de atenuación (a utilizar en los tipos de

radiación más intensa), etc. Si se detectaba algún error leve (como pueda ser un ligero desenfoque), el sistema paraba y se avisaba al operador que podía reanudar el tratamiento una vez corregida la situación de error. En caso de error grave (por ejemplo, una sobredosis de radiación) el sistema paraba por completo y debía ser reiniciado.

Este programa de control era una aplicación concurrente, compuesta por múltiples actividades, gestionando cada una de ella un conjunto diferente de elementos del sistema. Entre las actividades más importantes cabría distinguir:

- *Gestión de entrada.* Esta actividad monitorizaba la consola y recogía toda la información de configuración establecida por el operador. Se dejaba indicado en una variable global si el operador había finalizado su introducción de datos.
- *Gestión de los imanes.* Los componentes internos necesarios para la generación de la radiación necesitaban aproximadamente ocho segundos para posicionarse y aceptar la nueva especificación del tipo de radiación y su intensidad. Como el Therac-25 admitía dos tipos de radiación, en uno de ellos se utilizaban escudos de atenuación y en el otro no.
- *Pausa.* Existía una tercera actividad que suspendía la gestión de entrada mientras se estuviese ejecutando la gestión de los imanes. Con ello se intentaba que la gestión de los imanes pudiera finalizar lo antes posible. Se suponía que cualquier intento de reconfigurar el equipo durante esta pausa sería atendido cuando la pausa finalizara, justo antes de iniciar el tratamiento, obligando en ese punto a reiniciar la gestión de los imanes.

Lamentablemente, existía un error de programación en la aplicación concurrente y, además de suspender temporalmente la gestión de entrada, cuando finalizaba la actividad de "*pausa*" no se llegaba a consultar si el operador había solicitado una modificación de la configuración del sistema. Así, el operador observaba que la nueva configuración aparecía en pantalla (y suponía que había sido aceptada), pero el sistema seguía teniendo sus imanes adaptados a la configuración inicialmente introducida. Se había dado una "*condición de carrera*" y el estado real del sistema no concordaba con lo que aparecía en la pantalla del operador. Esto llegó a tener consecuencias fatídicas en algunos tratamientos (en ellos se llegó a dar la radiación de mayor intensidad sin la presencia de los escudos de atenuación): entre 1985 y 1987 se dieron seis accidentes de este tipo, causando tres muertes [LT93].

Este es uno de los ejemplos más relevantes de los problemas que puede llegar a causar una aplicación concurrente con errores en su diseño.

2. *Servidor web Apache* [Apa12]. Apache es un servidor web que emplea programación concurrente. El objetivo de un servidor web es la gestión de cierto conjunto de páginas web ubicadas en ese servidor. Dichas páginas resultan accesibles utilizando el protocolo HTTP (o *Hypertext Transfer Protocol*

[FGM+99]). Los navegadores deben realizar peticiones a estos servidores para obtener el contenido de las páginas que muestran en sus ventanas.

En un servidor de este tipo interesa tener múltiples hilos de ejecución, de manera que cada uno de ellos pueda atender a un cliente distinto, paralelizando así la gestión de múltiples clientes. Para que la propia gestión de los hilos no suponga un alto esfuerzo, Apache utiliza un conjunto de hilos (o "*thread pool*") en espera, coordinados por un hilo adicional que atiende inicialmente las peticiones que vayan llegando, asociando cada una de ellas a los hilos del conjunto que queden disponibles. Al tener ese conjunto de hilos ya generados y listos para su asignación se reduce el tiempo necesario para iniciar el servicio de cada petición. Si el número de peticiones recibidas durante cierto intervalo de tiempo supera el tamaño de ese conjunto, las peticiones que no pueden servirse de inmediato se mantienen en una cola de entrada. El objetivo de este modelo de gestión es la reducción del tiempo necesario para gestionar a los hilos que sirvan las peticiones recibidas. De hecho, los hilos ya están creados antes de iniciar la atención de las peticiones y no se destruyen cuando finalizan la atención de cada petición, sino que vuelven al "*pool*" y pueden ser reutilizados posteriormente. Con ello, en lugar de crear hilos al recibir las peticiones o destruirlos al finalizar su servicio (dos operaciones que requieren bastante tiempo) basta con asignarlos o devolverlos al "pool" (operaciones mucho más rápidas que las anteriores).

3. *Videojuegos.* La mayoría de los videojuegos actuales (tanto para ordenadores personales como para algunas consolas) se estructuran como aplicaciones concurrentes compuestas por múltiples hilos de ejecución. Para ello se suele utilizar un *motor de videojuegos* que se encarga de proporcionar cierto soporte básico para la renderización, la detección de colisiones, el sonido, las animaciones, etc. Los motores permiten que múltiples hilos se encarguen de estas facetas. Así, el rendimiento de los videojuegos puede aumentar en caso de disponer de un equipo cuyo procesador disponga de múltiples núcleos.

   Estos motores también facilitan la portabilidad del juego, pues proporcionan una interfaz que no depende del procesador ni de la tarjeta gráfica. Si el motor ha llegado a implantarse sobre múltiples plataformas, los juegos desarrollados con él se pueden migrar a ellas sin excesivas dificultades. Por ejemplo, la detección de colisiones en un motor de videojuegos suele ser responsabilidad de un componente llamado *motor de física* ("*physics engine*"). En el juego FIFA de Electronic Arts, este motor (que en este caso se llama "*Player Impact Engine*") se renovó en la segunda mitad de 2011. El motor de videojuegos del que forma parte ha facilitado su portabilidad a algunas de las plataformas en las que FIFA 12 está disponible: Xbox 360, Wii, Nintendo 3DS, PC, PlayStation 3, PlayStation 2, PlayStation Portable, PlayStation Vita, Mac, iPhone, iPad, iPod y Android.

4. *Navegador web.* Cuando Google presentó su navegador *Chrome* (en diciembre de 2008), su arquitectura [McC08] era muy distinta a la del resto de navegadores web (Microsoft Internet Explorer, Mozilla Firefox, Opera, ...). En Chrome, cada pestaña abierta está soportada por un proceso independiente. De esa manera, si alguna de las pestañas falla, el resto de ellas puede seguir sin problemas. Para mejorar su rendimiento se optimizó su soporte para Javascript, compilando y manteniendo el código generado en lugar de interpretarlo cada vez. Además, en cada pestaña se utilizan múltiples hilos para obtener cada uno de los elementos de la página que deba visualizarse. Con ello, la carga de una página podía realizarse de manera mucho más rápida que en otros navegadores. Actualmente la mayor parte de los navegadores web han adoptado una arquitectura similar a la de Chrome, dadas las ventajas en rendimiento y robustez que proporciona.

1.3.3 Problemas clásicos

Existe una serie de problemas clásicos (o académicos) de la programación concurrente que ilustran una serie de situaciones en las que múltiples actividades deben seguir ciertas reglas para gestionar un recurso compartido de manera adecuada.

Aunque se incluyan en esta sección, los ejemplos que describiremos seguidamente se utilizarán en las próximas unidades para ilustrar algunos de los problemas que se plantean a la hora de desarrollar una aplicación concurrente. Por ello, no es necesario que se estudien con detenimiento dentro de esta primera unidad didáctica, pero sí que se revisen más tarde en una segunda lectura al analizar esas unidades didácticas posteriores.

Productor-Consumidor con buffer acotado

En este problema [Hoa74] se asume que existen dos tipos de actividades o procesos: los productores y los consumidores. La interacción entre ambos roles se lleva a cabo utilizando un *buffer* de capacidad limitada, en el que los procesos *productores* irán depositando elementos y de donde los procesos *consumidores* los extraerán. Este buffer gestiona sus elementos con una estrategia *FIFO* ("*First In, First Out*"), es decir, sus elementos se extraerán en el orden en que fueron insertados.

Para que el uso del buffer sea correcto, tendrá que respetar una serie de restricciones que discutiremos en función del ejemplo mostrado en la figura 1.1. En esta figura se observa un buffer con capacidad para siete elementos, construido como un vector cuyas componentes se identifican con los números naturales entre 0 y 6, ambos inclusive. Para gestionar este buffer utilizaríamos (en cualquier lenguaje de programación orientado a objetos) estos atributos:

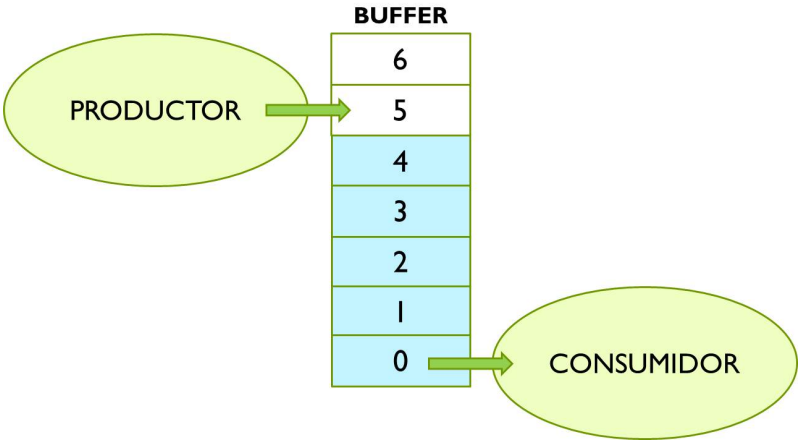

Figura 1.1: Uso de un buffer de capacidad limitada.

- `in`: Indicaría la componente en la que algún proceso productor insertaría el próximo elemento en el buffer. En el estado mostrado en la figura 1.1, su valor sería 5.
- `out`: Indicaría qué componente mantiene el elemento que algún consumidor extraerá a continuación. En la figura 1.1 su valor sería 0.
- `size`: Mantendría el tamaño del buffer. En este ejemplo sería 7.
- `numElems`: Mantendría el número actual de elementos en el buffer. En la figura 1.1 su valor sería 5.
- `store`: Vector en el que se almacenan los elementos del buffer.

Para que un productor inserte un elemento en el buffer, se tendrá que seguir un algoritmo similar al siguiente:

1. Insertar el elemento en la posición `in` del vector `store`.
2. Incrementar el valor de `in` de manera circular. Es decir: `in` ← `(in + 1) MOD size`.
3. Incrementar el número de elementos en el buffer. Es decir: `numElems++`.

Por su parte, para que un consumidor extraiga un elemento del buffer, se utilizaría este algoritmo:

1. Retornar el elemento ubicado en la posición `out` del vector `store`.

2. Incrementar el valor de `out` de manera circular. Es decir: `out` ← `(out + 1) MOD size`.

3. Decrementar el número de elementos en el buffer. Es decir: `numElems--`.

Estos algoritmos básicos podrían ocasionar problemas al ser ejecutados de manera concurrente. Veamos algunos ejemplos:

- Si en la situación mostrada en la figura 1.1 dos procesos productores iniciaran a la vez la inserción de un nuevo elemento en el buffer, ambos observarían en el primer paso de su algoritmo el mismo valor para el atributo `in`. A causa de ello, los dos insertarían sus elementos en la posición 5 y así el segundo elemento insertado sobrescribiría al primero. Posteriormente, ambos incrementarían el valor de `in` (que pasaría primero a valer 6 y después a valer 0), así como el valor de `numElems`, que llegaría a 7. Sin embargo, no habría ningún elemento en la componente 6 del vector `store`.

- De manera similar, si en esa misma situación inicial dos procesos consumidores trataran de extraer de manera simultánea un elemento del buffer, probablemente ambos obtendrían el elemento contenido en la componente 0 del vector `store`. Posteriormente se incrementaría el valor de `out` (hasta llegar a 2) y se decrementaría el valor de `numElems` que pasaría a valer 3. Cada uno de ellos estaría convencido de haber obtenido un elemento distinto, pero en lugar de ello, cada uno estaría procesando por su cuenta el mismo elemento. Por contra, el elemento mantenido en la componente 1 jamás llegaría a ser utilizado por ningún consumidor y sería sobrescrito posteriormente por algún elemento insertado por un productor.

- Si a partir de la situación mostrada en la figura 1.1 tres procesos productores llegasen a insertar elementos sin que ningún consumidor extrajera ninguno mientras tanto, la capacidad del buffer se desbordaría. Debido a ello, la tercera inserción sobrescribiría el elemento mantenido en la posición 0 del vector `store`.

- Si a partir de la situación de la figura 1.1 seis procesos consumidores extrajeran elementos del buffer, sin que ningún productor insertase algún nuevo elemento, el último de ellos encontraría un buffer vacío. Sin embargo, no sería consciente de ello y retornaría el contenido de la posición 5 del vector.

Para evitar estas situaciones incorrectas, deberían imponerse y respetarse las siguientes restricciones:

PC1 Para evitar la primera y la segunda situación de error se debería impedir que los métodos de acceso al buffer fuesen ejecutados por más de una actividad simultáneamente. Es decir, cuando alguna actividad haya iniciado la ejecución de algún método, ninguna actividad más podrá ejecutar alguno de los métodos de acceso al buffer.

PC2 Para evitar que se desbordara el buffer se debería impedir que los productores ejecutasen el método de inserción cuando el valor de los atributos `size` y `numElems` fueran iguales.

PC3 Para evitar que se extrajeran elementos de un buffer vacío se debería impedir que los consumidores ejecutasen el método de extracción cuando el atributo `numElems` ya valiese cero.

Lectores-Escritores

En este problema [CHP71] se asume que existe un recurso compartido por todas las actividades. El estado de este recurso se puede modificar. Algunas actividades utilizarán métodos para modificar el estado del recurso: se considerarán "*escritores*". Otras actividades o procesos llegarán a leer el estado del recurso, pero no lo modificarán: son "*lectores*". Un ejemplo válido de recurso de este tipo sería un fichero, que podrá ser leído o escrito por diferentes procesos.

Para garantizar un uso correcto del recurso se imponen las siguientes restricciones:

LE1 Múltiples procesos lectores pueden acceder simultáneamente al recurso.

LE2 Sólo un proceso escritor puede estar accediendo al recurso. En caso de que un escritor acceda, ningún proceso más podrá utilizar el recurso.

Así se garantiza la consistencia en las modificaciones realizadas sobre el recurso. Mientras un proceso escriba sobre él ninguna escritura más llegará a causar interferencias. También se evita que algún proceso lector pudiera leer un estado intermedio durante la escritura.

Cinco filósofos

En este problema [Dij71, Hoa85] se asume que existen cinco filósofos cuya única misión es pensar y comer. En la mesa que comparten hay cinco plazas. A estos filósofos solo les gustan los *spaghetti* y para comer necesitan dos tenedores cada uno. Sin embargo, en la mesa hay cinco platos y cinco tenedores, tal como se muestra en la figura 1.2. Por tanto, cuando un filósofo vaya a comer tendrá que coger los dos tenedores que queden a ambos lados de su plato, pero para ello ninguno de sus dos filósofos vecinos podrá estar comiendo.

Este ejemplo ilustra el problema de la escasez de recursos en un sistema, pues no hay suficientes tenedores para que todos los filósofos coman a la vez. De hecho, en el mejor de los casos solo habrá dos filósofos comiendo.

El algoritmo utilizado por cada filósofo sigue estos pasos:

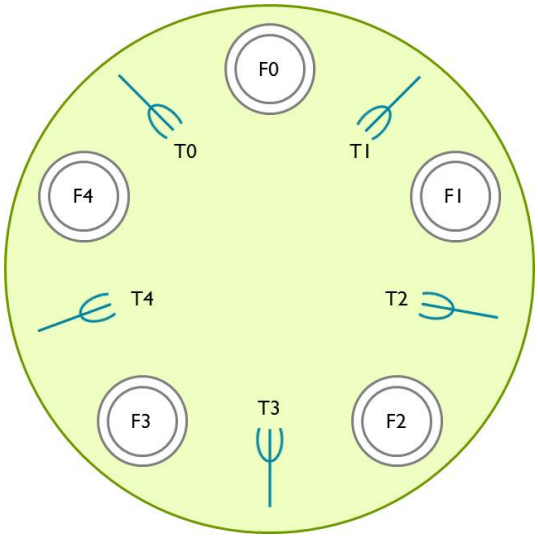

Figura 1.2: Problema de los cinco filósofos.

1. Coger el tenedor derecho.
2. Coger el tenedor izquierdo.
3. Comer.
4. Dejar ambos tenedores.
5. Pensar.
6. Volver al paso 1 cuando tenga hambre.

Los tenedores no pueden compartirse. En todo momento un tenedor estará libre o asignado a un solo filósofo, pero nunca a dos simultáneamente.

Si un filósofo observa que el tenedor que necesitaba en el paso 1 o en el paso 2 está asignado a su filósofo vecino, espera educadamente a que éste termine y lo libere.

Si todos los filósofos empezaran a la vez con sus respectivos algoritmos podrían obtener todos ellos su tenedor derecho. Sin embargo, al intentar obtener el tenedor izquierdo, ninguno de ellos lo conseguiría. Todos observarían que dicho tenedor ya está asignado a su vecino. Así, todos ellos pasarían a esperar a que tal tenedor quedara libre, pero eso nunca sucedería. Esta situación se conoce como *interbloqueo* [CES71] y se explicará con detenimiento en la unidad didáctica 4.

1.4 Tecnología Java

Para poder aprender qué es la concurrencia se necesita implantar algunos ejemplos de programas concurrentes. Para ello debe seleccionarse algún lenguaje de programación que soporte directamente el uso de múltiples hilos de ejecución en un programa. Se ha seleccionado *Java* para cubrir este objetivo por múltiples razones:

- Es un lenguaje relativamente moderno y ampliamente aceptado.
- Proporciona un soporte adecuado tanto para la programación concurrente como para la programación distribuida.
- Facilita un amplio conjunto de herramientas de sincronización que permitirán implantar de forma sencilla los conceptos que se analizan en este libro relacionados con la sincronización.
- Es independiente de la plataforma. Utiliza una *máquina virtual* [LYBB11] propia para ejecutar los programas que está soportada en la mayoría de los sistemas informáticos actuales. Con ello, el código que se obtiene al compilar un programa no depende para nada del sistema operativo utilizado en el ordenador donde ejecutaremos el programa.
- Existe abundante documentación sobre este lenguaje de programación. Buena parte de ella es gratuita. Existen guías y tutoriales [Ora12b] disponibles en la web de *Oracle* [Ora12e].

1.4.1 Concurrencia en Java

Java soporta nativamente (es decir, sin necesidad de importar ninguna biblioteca o "*package*" adicional) la creación y gestión de múltiples hilos de ejecución en todo programa que se escriba en este lenguaje. Aparte de los hilos que podamos crear explícitamente a la hora de implantar un determinado programa, todo proceso Java utilizará algunos hilos de ejecución implícitos (utilizados para gestionar la interfaz de usuario, en caso de utilizar una interfaz gráfica basada en ventanas, y para gestionar la *recolección de residuos* [WK00, Appendix A], es decir, para realizar la eliminación de aquellos objetos que ya no estén referenciados por ningún componente de la aplicación).

Se debe recordar que un *hilo de ejecución* (o "*thread*") es la unidad de planificación comúnmente utilizada en los sistemas operativos modernos. Por su parte, un *proceso* es la entidad a la que se asignan los recursos y mantiene a un programa en ejecución. Por tanto, cada proceso tendrá un *espacio de direcciones* independiente, mientras que todos los hilos de ejecución creados en una misma ejecución de un

programa determinado compartirán ese espacio de direcciones asignado al proceso que los engloba.

En el caso particular de Java, un conjunto de clases podrá definir una aplicación. En una de tales clases deberá existir un método `main()` que será el que definirá el hilo principal de ejecución de esa aplicación. Para poder ejecutar la aplicación habrá que utilizar una máquina virtual Java (o JVM) [LYBB11] y esa acción implicará la creación de un proceso soportado por el sistema operativo utilizado en ese ordenador.

1.4.2 Gestión de hilos de ejecución

La gestión básica de los hilos de ejecución en Java será analizada en la unidad 2. No obstante, se describe seguidamente cómo se pueden definir hilos en un programa Java y qué debe hacerse para asignarles un identificador y averiguar la identidad de cada hilo.

Creación de hilos

Para definir un hilo de ejecución en Java debemos definir alguna instancia de una clase que implante la interfaz `Runnable`. Java ya proporciona la clase `Thread` que implanta dicha interfaz. Por tanto, disponemos de cuatro alternativas básicas para definir hilos, tal como se muestra en la Tabla 1.1.

	Clase con nombre	**Clase anónima**

Implantando `Runnable`	public class H implements Runnable {
	  public void run() {
	  System.out.println("Ejemplo.");
	  }
	}
	Thread t = new Thread(new H());
		Runnable r = new Runnable() {
		  public void run() {
		  System.out.println("Ejemplo.");
		  }
		};
		Thread t = new Thread(r);

Extendiendo `Thread`	public class H extends Thread {
	  public void run() {
	  System.out.println("Ejemplo.");
	  }
	}
	H t = new H();
		Thread t = new Thread() {
		  public void run() {
		  System.out.println("Ejemplo.");
		  }
		};

Tabla 1.1: Variantes de la definición de hilos.

Obsérvese que el código que tendrá que ejecutar el hilo debe incluirse en el método `run()`. En la Tabla 1.1 hemos incluido una sola sentencia en dicho método: la necesaria para escribir la palabra "Ejemplo" en la salida estándar.

La columna izquierda muestra aquellas variantes en las que se llega a definir una clase (a la que se ha llamado `H`) para generar los hilos. Cuando se adopta esta

opción será posible generar múltiples hilos de esa misma clase, utilizando una sentencia "`new H()`". Por su parte la columna derecha ilustra el caso en que no se necesite generar múltiples hilos y baste con definir directamente aquél que vaya a utilizarse. En estos ejemplos la variable utilizada para ello ha sido "`t`".

A su vez, la fila superior muestra cómo debe generarse el hilo cuando se está implantando directamente la interfaz `Runnable`. Si definimos una clase (columna izquierda) habrá que añadir en su declaración un "`implements Runnable`" antes de abrir la llave que define el bloque en el que se definirán sus atributos y métodos. Por su parte, la columna derecha ilustra cómo se puede definir una instancia de dicha interfaz (utilizando para ello la variable "`r`") que después podrá utilizarse como argumento en el constructor del hilo "`t`". En este caso el código del método "`run()`" debe incluirse al definir "`r`".

La fila inferior muestra cómo debe realizarse esta gestión en caso de generar una subclase de `Thread` (si utilizamos una clase explícita, en la columna izquierda) o cómo instanciar un `Thread` (en caso de que solo interese generar un hilo, en la columna derecha) con su propio código para "`run()`".

Obsérvese que tras utilizar el código mostrado en la Tabla 1.1 en cualquiera de sus cuatro variantes se habrá generado un hilo Java, cuya referencia mantendremos en la variable "`t`". Este hilo todavía no se ejecutará, pues lo que hemos conseguido con estas definiciones es simplemente generarlo, de manera que el *runtime* de Java le habrá asignado todos los recursos necesarios para su ejecución, pero su estado de planificación será todavía "*nuevo*" en lugar de "*preparado*". Para que un hilo pase a estado "*preparado*" y así pueda iniciar su ejecución cuando el planificador del sistema operativo lo seleccione, debe utilizarse el método "`start()`".

Por ello, en alguna de las líneas que sigan a las mostradas en la Tabla 1.1 deberíamos encontrar una sentencia "`t.start();`". Esa sentencia inicia la ejecución del hilo referenciado por la variable "`t`".

Aunque la sentencia "`t.run();`" parece sugerir un efecto similar y no genera ningún error ni excepción, su resultado es muy diferente. Si dicha sentencia es ejecutada por un hilo A, en lugar de iniciarse la ejecución del hilo "`t`", lo que ocurrirá es que A será quien ejecute las sentencias del método "`run()`" y "`t`" seguirá todavía en el estado "*nuevo*".

Identificación de hilos

Al crear un hilo es posible asociarle un nombre. La clase `Thread` admite como argumento en su constructor una cadena que será la utilizada como nombre o identificador del hilo que se genere.

Si se necesitara modificar el nombre de un hilo una vez ya se haya generado éste, se puede utilizar el método `setName()`. A su vez, la clase `Thread` ofrece un método `getName()` para obtener el nombre de un hilo (aunque necesitamos una variable que lo referencie). Así, si queremos escribir el nombre del hilo actualmente en ejecución tendremos que utilizar esta sentencia:

```
System.out.println( Thread.currentThread().getName() );
```

La figura 1.3 lista un programa Java que utiliza los métodos explicados en esta unidad para lanzar 10 hilos de ejecución, que se encargan de escribir su propio nombre para finalizar de inmediato. Cada uno de esos hilos recibe como identificador la cadena "ThreadX" siendo X cada uno de los dígitos entre 0 y 9, ambos inclusive.

```
public class Sample {
    public static void main( String args[ ] ) {
        System.out.println(Thread.currentThread().getName());
        for(int i=0; i<10; i++) {
            new Thread("Thread"+i) {
                public void run() {
                    System.out.println("Executed by " +
                        Thread.currentThread().getName() );
                }
            }.start();
        }
    }
}
```

Figura 1.3: Ejemplo de creación de hilos.

Obsérvese que para asignar los nombres a los hilos se ha empleado el propio constructor de la clase `Thread` y para imprimir el nombre de cada hilo se ha utilizado el método `getName()`. Por último, ya que todos los hilos se estaban generando en una misma sentencia dentro del bucle "`for`" se ha podido utilizar la variante anónima de creación de hilos. Por ello, se ha añadido una llamada al método `start()` tras la llave que cerraba la definición de la clase.

1.5 Resumen

En esta unidad didáctica se han introducido los conceptos básicos de la programación concurrente. Un programa concurrente está constituido por diferentes actividades, que pueden ejecutarse de forma independiente, las cuales se reparten la solución del problema. Para ello, estas actividades trabajan en común, coordinándose y comunicándose entre sí empleando variables en memoria y mensajes, para así compartir datos y proporcionarse sus resultados. La programación concurrente permite mejorar las prestaciones del sistema, aumentando su eficiencia y escalabilidad, proporcionando también una gestión eficiente de las comunicaciones. Además, la programación concurrente mejora la interactividad y flexibilidad de las tareas, al utilizar diseños modulares y estructurados. Asimismo, en muchas ocasiones las aplicaciones requieren el uso de varias actividades simultáneas, por lo que resulta más directo diseñarlas a través de la programación concurrente (ya que es menor el hueco semántico que se produce), que si se intenta diseñarlas como un solo programa secuencial.

A lo largo de esta unidad didáctica se han presentado diferentes aplicaciones concurrentes reales, tales como el programa de control de acelerador lineal Therac-25 y el servidor web Apache, así como una serie de problemas clásicos (o académicos) de la programación concurrente, con los que se ilustran las características de la programación concurrente y los problemas y dificultades que se plantean a la hora de desarrollar este tipo de aplicaciones. Para dar soporte a los ejemplos de programas concurrentes, se ha seleccionado el lenguaje de programación Java, que permite la creación y gestión de múltiples hilos de ejecución. Precisamente, en esta unidad se describe cómo se definen hilos en Java y cómo asignarles un identificador.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Discriminar entre programación secuencial y programación concurrente.
- Describir las ventajas e inconvenientes que proporciona la programación concurrente.
- Enumerar distintas aplicaciones concurrentes, tanto aplicaciones reales como problemas clásicos.
- Implementar hilos en Java.
- Identificar las secciones de una aplicación que deban o puedan ser ejecutadas concurrentemente por diferentes actividades.

Para seguir leyendo haga click aquí