UD3: Programación Multihilo

PSP

Ana Alonso

Curso 2024-2025

Programación multihilo

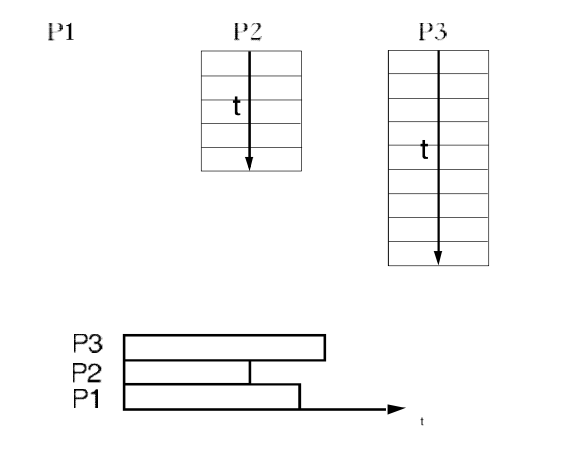
- Objetivos
  - Desarrollar aplicaciones multihilo.
  - Desarrollar aplicaciones con varios hilos que deban sincronizarse entre ellas, intercambiando información.
  - Desarrollar aplicaciones que creen y ejecuten varios hilos sincronizandos gestionando su prioridad.
- Contenido:
  - Presentación de la problemática.
  - Sección crítica.
  - Herramientas de sincronización de procesos.
  - Herramientas de comunicación de procesos.
  - Interbloqueo.
  - Herramientas de concurrencia en Java.

Ejecución concurrente

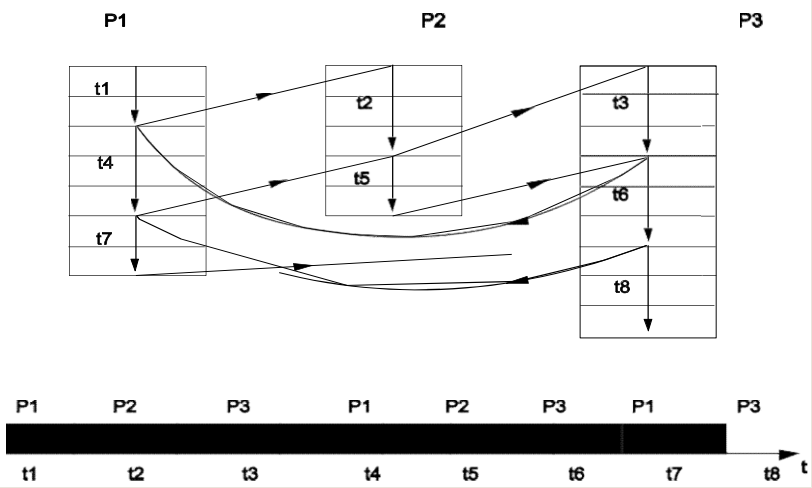
Se dice que dos procesos son concurrentes cuando su ejecución se simultanea en el tiempo

- En sistemas monoprocesador, la ejecución concurrente es aparente y se produce al intercalar los procesos en los cambios de contexto ("paralelismo virtual")
- En sistemas multiprocesador, puede darse verdadera ejecución concurrente al disponer de varias CPU

Punto de vista del usuario



Ejecución real



Aplicaciones concurrentes

- Son aquellas que descomponen el trabajo en tareas que se pueden ejecutar concurrentemente
- Para que el resultado final sea correcto, se requiere garantizar:
  - La consistencia e integridad de cada tarea y los datos que maneja
  - Generalmente, un orden parcial de ejecución entre tareas
- El control de concurrencia se realiza con diversas herramientas y puede ser:
  - Explícito: el programador tiene que encargarse de él
  - Implícito: el núcleo o las bibliotecas del sistema lo realizan de manera transparente para el programador

Recurso compartido

- Es todo aquel recurso que puede ser utilizado concurrentemente por dos o más procesos o hilos
- Ejemplos:
  - Una estructura de datos en memoria compartida
  - Un fichero
  - Un periférico

Ejemplos de concurrencia

- Uso de recursos compartidos:
  - Generar identificadores únicos
  - Modificar un fichero compartido
  - Enviar datos a la impresora
  - Pedir más memoria

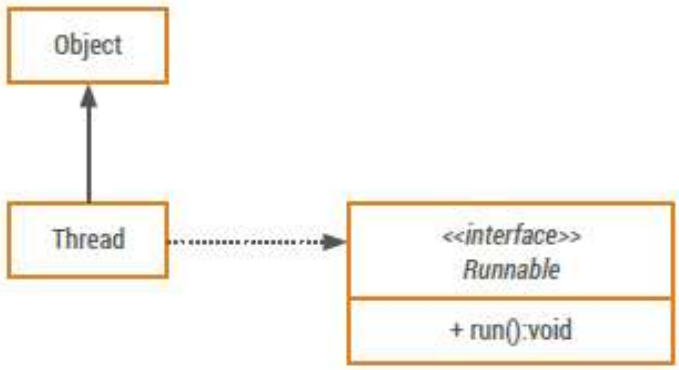
- Secuenciación de operaciones:
  - Un proceso lee pistas de audio y otro las va codificando después de leerlas
  - Una simulación reparte el cálculo en varios procesos paralelos y cuando todos terminan tiene que integrar los resultados parciales

Multitarea en Java I

La programación multitarea en Java consiste en organizar un programa para que realice más de una tarea a la vez, dividir el programa en subprocesos que pueden ejecutarse concurrentemente si se dispone de más de un procesador o compartiendo el tiempo del procesador.

En Java los **subprocesos** en los que se puede organizar un programa se llaman **"threads"** Un **"thread"** es un flujo de control secuencial dentro de un programa. Los programas vistos hasta este momento no son multitarea, sólo tienen un flujo de control, sólo existe un único **"thread"**.
Todos los programas tienen al menos un subproceso de ejecución, es el llamado subproceso principal, es el que se ejecuta al iniciarse el programa. Hasta este momento es el único que se ha utilizado. A partir de este subproceso principal se crean los demás.

Multitarea en Java II



La implementación en Java de la multitarea basada en procesos se basa en la clase **Thread** y en la interfaz **Runnable**.

En el método **run** es donde tiene lugar la acción del subproceso, es donde se define el código que ejecutará el subproceso. Dentro del método run puede invocar otras funciones, usar otras clases y declarar variables al igual que el subproceso principal.

La clase Thread y la interfaz Runnable

Hay dos modos de proporcionar el método run a un subproceso:

- En una subclase de Thread sobrecargandoel método run.
- En cualquier clase implementando el interface Runnable y
- definiendo su función run.

La razón de que existan estas dos posibilidades es porque en Java no existe la herencia múltiple. Si una clase deriva de otra no podemos hacer que derive también de Thread.

Algunas operaciones con hilos en Java

```
Thread hilo;
```

- Punto de inicio del hilo:

```
hilo.run();
```

- Arrancar un hilo:

```
hilo.start();
```

- Esperar a que termine un hilo:

```
hilo.join();
```

- Hacer que el hilo actual se bloquee durante un tiempo:

```
Thread.sleep(milisegundos);
```

- Desbloquear un hilo que estuviera bloqueado:

```
hilo.interrupt();
```

- Ver API de java.lang.Thread

Subprocesos. Ciclo de vida

Un subproceso desde que se crea como objeto “Runnable”, hasta que se libera el espacio que ocupa en
memoria, pasa por una serie de estados. Los estados de un subproceso son los que están indicados en la enumeración Thread.State. La función static getState() de Thread devuelve el estado actual del
subproceso en ejecución.

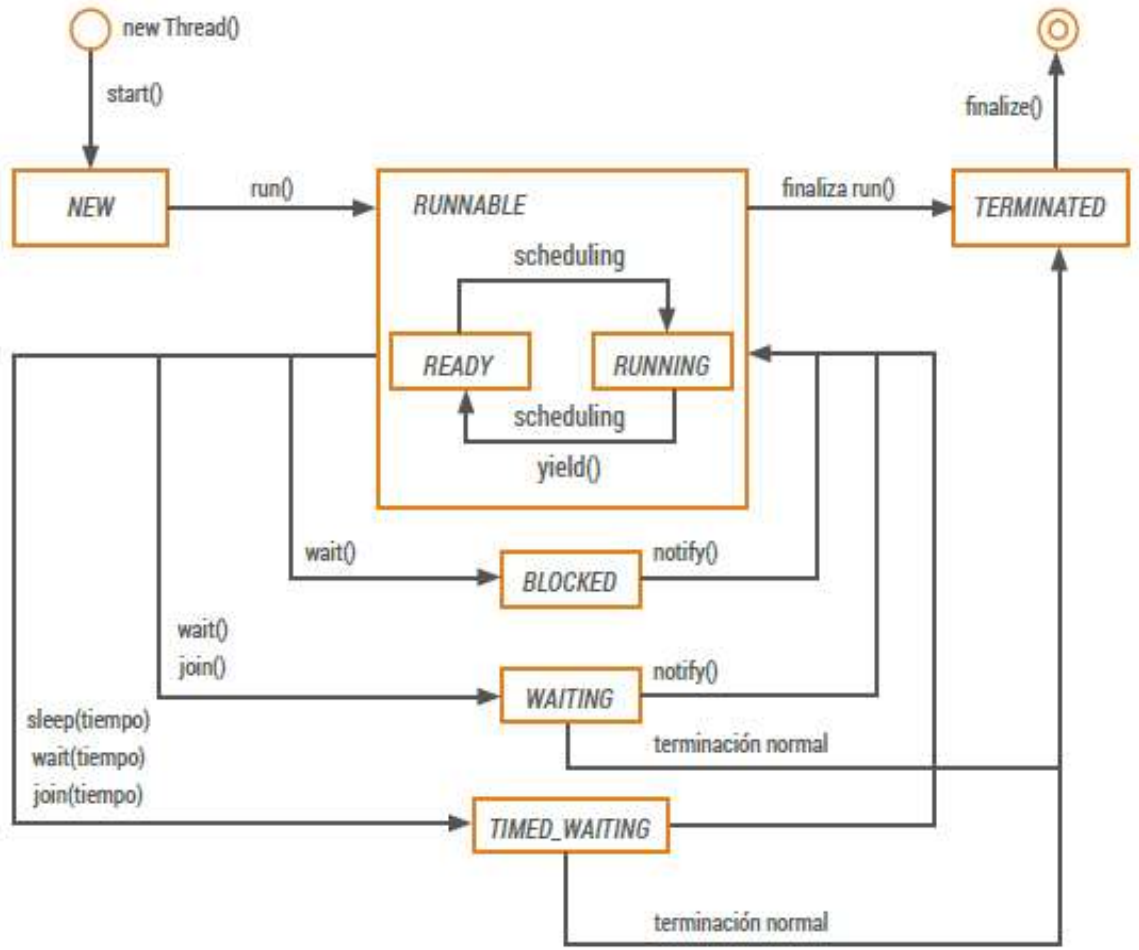
Subprocesos. Ciclo de vida

.

Los estados de un proceso son:

Estado	Funcionalidad
NEW	Se ha creado el subproceso pero todavía no ha arrancado.
RUNNABLE	El subproceso se esta ejecutando en la JVM. De acuerdo con el planificador de tareas (scheduler) del sistema operativo esta en ejecución o preparado para seguir ejecutándose, de acuerdo a su prioridad, ocupando un "slice" de tiempo del procesador. El planificador de tareas tiene que asegurar que todos los subprocesos en este estado tienen que llegar a ejecutarse.
BLOCKED	El subproceso se encuentra bloqueado, esperando a volver a ejecución.
WAITING	El subproceso se encuentra esperando indefinidamente, a que otro realice una determinada acción que lo saque de este estado.
TIMED_ WAITING	El subproceso se encuentra esperando durante un tiempo determinado.
TERMINATED	El subproceso ha finalizado su ejecución, ha terminado la función run.

Subprocesos. Ciclo de vida



Problema de concurrencia

Dos hilos comparten una variable en memoria

Instrucciones en lenguaje de alto nivel

```
int suma;


void hilo1 (void){
suma = suma + 3;
}


void hilo2 (void){
suma = suma + 5;
}
```

Descomposición de la instrucción `suma=suma+valor` en lenguaje de ensamble que se ejecuta concurrentemente por los dos hilos (10000100 es la dirección en memoria de la variable suma)

```
LDR R0,=0x10000100
LDR R1,[R0]
MOV R2,#3
ADD R1,R2
STR R1,[R0]
```

```
LDR R0,=0x10000100
LDR R1,[R0]
MOV R2,#5
ADD R1,R2
STR R1,[R0]
```

Sección crítica

- Sección de código de un proceso o hilo que accede a unos recursos compartidos y, mientras los usa, otros procesos o hilos no deberían acceder a esos mismos recursos.

- Para conseguirlo las secciones críticas deberán ejecutarse **atómicamente**.
- Una operación atómica es aquella en la que otros procesos (o hilos) no ven los estados intermedios de la operación.
- Es crucial identificar adecuadamente estas secciones críticas y que estas sean **mínimas** para que exista más concurrencia.

Solución al problema de la sección crítica

Requisitos que debe ofrecer cualquier solución para resolver el problema de la sección crítica:

- Exclusión mutua:

  Si un proceso está ejecutando código de la sección crítica, ningún otro proceso lo podrá hacer.

- Progresión:

  Un proceso permanece en la sección crítica un tiempo finito.

- Espera limitada:

  Cuando un proceso quiera entrar en una sección crítica, se le autorizará en un tiempo finito.

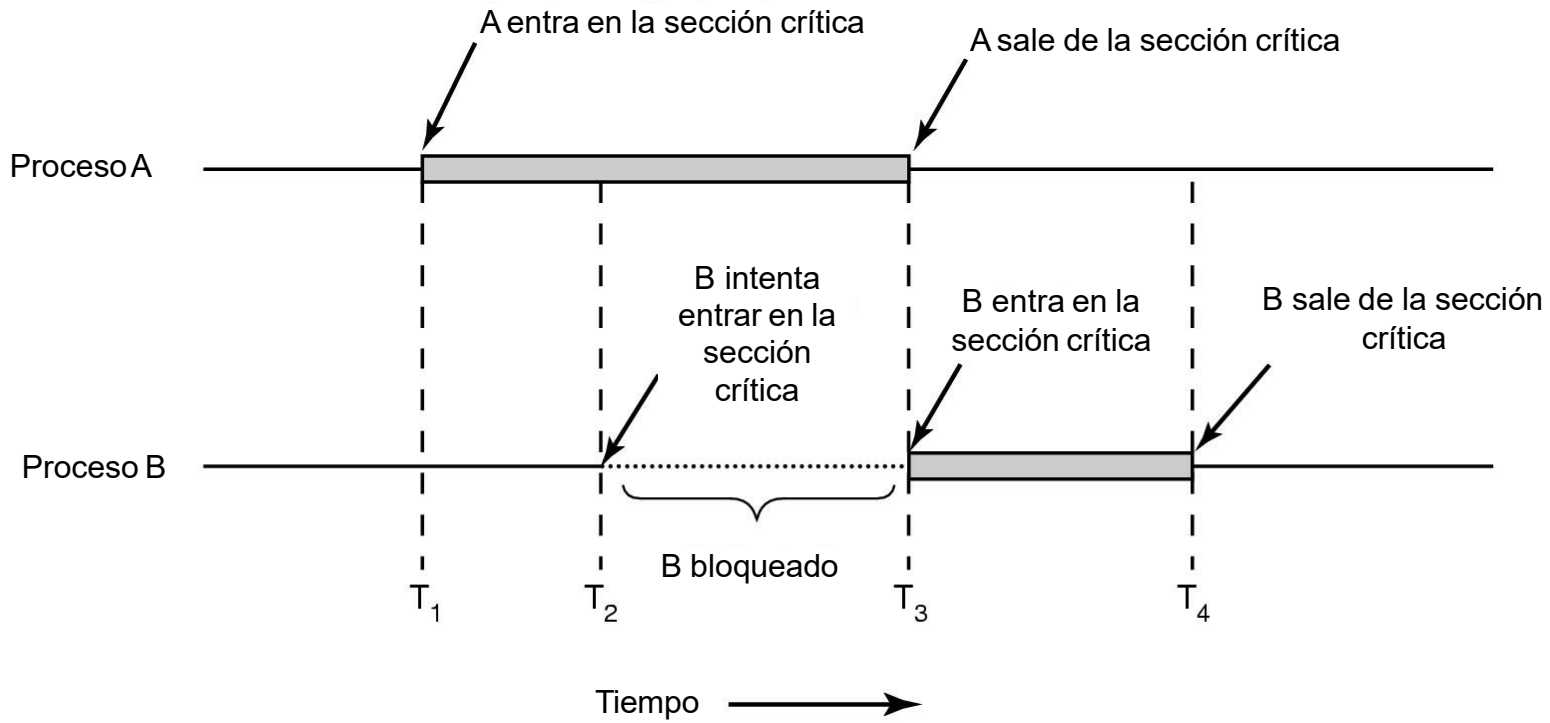
Solución al problema de la sección crítica

Estructura general de cualquier mecanismo utilizado para resolver el problema de la sección crítica:

Entrada en la sección crítica

Código de la sección crítica

Salida de la sección crítica



Región crítica

- Definida por Brinch Hansen (1972) como **herramienta de programación** para resolver el problema de la sección crítica.
- Sintaxis

Region V **do**

```
Sección crítica sobre V
```

End Region

- Implementación en java (V es un objeto)

```
synchronized (V) {
   Sección crítica sobre V
}
```

Región Crítica Condicional

- Definida por Brinch Hansen como **herramienta de programación** para poder esperar a que se cumpla una condición dentro de una región crítica liberando la RC durante la espera.
- Sintaxis

Region V **when** B **do**

```
  Sección crítica sobre V
```

End Region

- Implementación en java

```
synchronized(V) {  while
  (!B) V.wait();
  Sección crítica sobre V
  V.notify/notifyAll();
}
```

Monitores (Hoare)

- Herramienta de sincronización que permite a los hilos:
  - Ejecutar las operaciones sobre el monitor en exclusión mutua.
  - Esperar de manera inactiva a que se cumpla una condición.
  - Avisar a los demás hilos cuando se ha producido una condición.
    - En java, cualquier clase puede ser usada como monitor:
      - Declarando **synchronized** todos los métodos públicos.
  - Esperar a que se cumpla una condición B:
    - while (! B) wait();
  - Informar a los demás hilos de que su condición puede haber cambiado
    - notify();
    - notifyAll();

Mutex o cerrojo

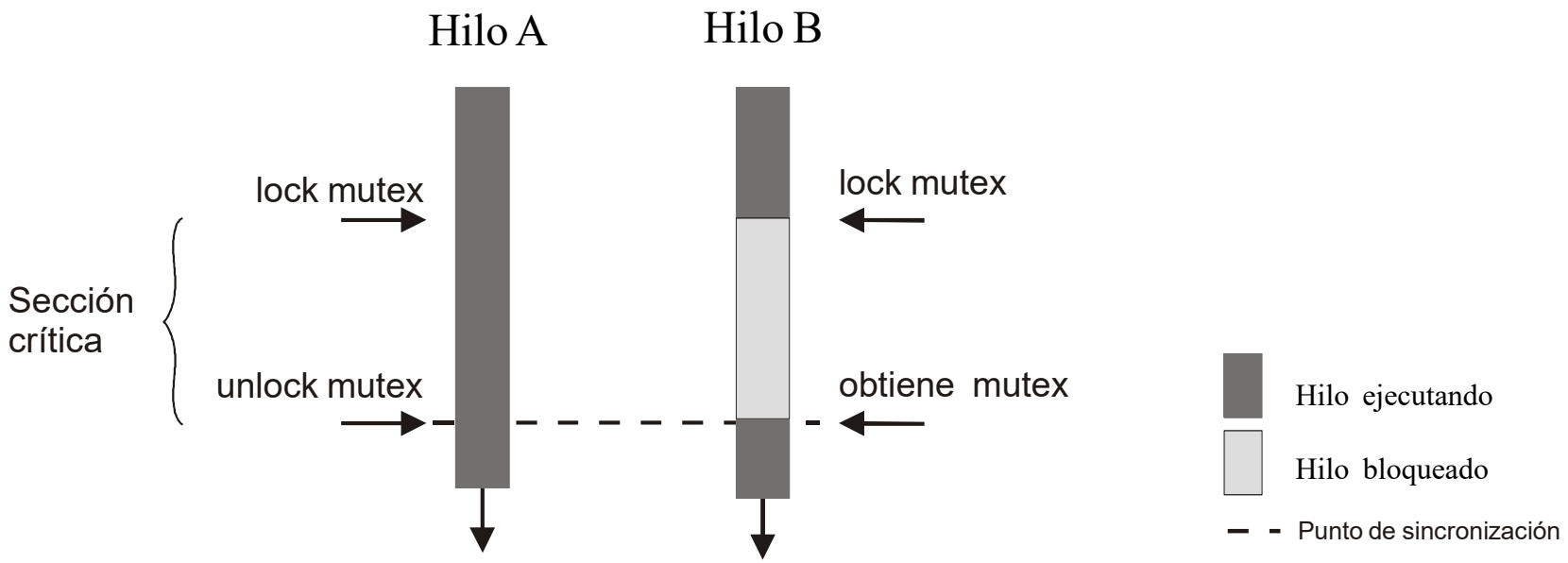
- Un mutex (o cerrojo) es una herramienta de sincronización que sirve para proteger secciones críticas que trabajan con más de un recurso compartido.
- Tiene dos operaciones atómicas:
  - **lock()** Si el cerrojo está abierto, lo cierra. En otro caso, se bloquea.
  - **unlock()** Si existen hilos bloqueados en el cerrojo, se lo transfiere a uno y lo desbloquea. En otro caso, deja abierto el cerrojo.

Mutex o cerrojo

- Implementación de la región crítica:

```
lock();     /* entrada en la región critica */
< seccion critica >
unlock();   /* salida de la región critica */
```

- La operación `unlock()` debe realizarla el hilo que ejecutó `lock()`



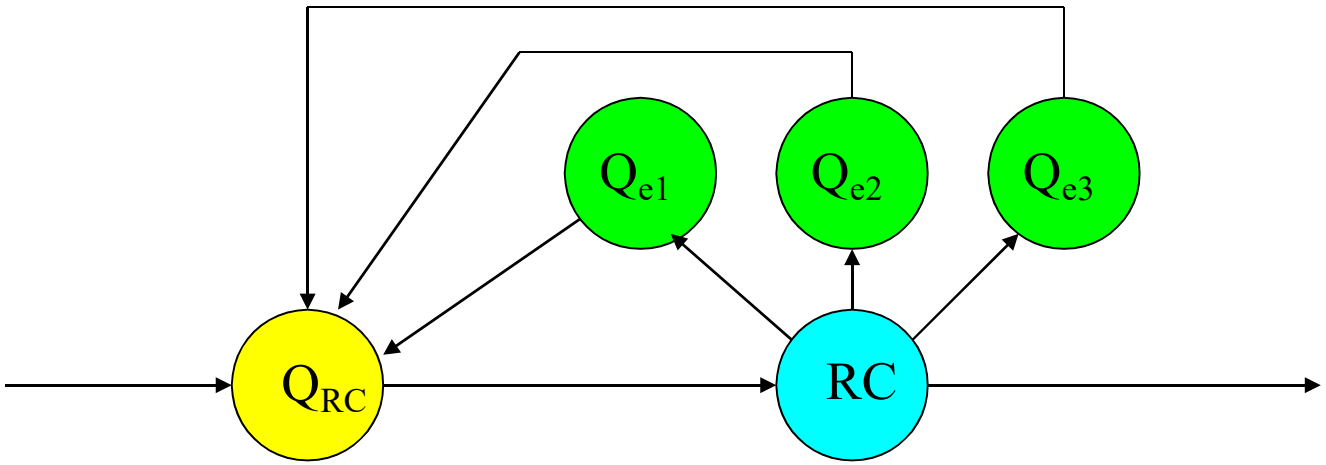
Cerrojos en Java

Implementación en java

```
Lock cerrojo;
...
cerrojo = new ReentrantLock();
...
cerrojo.lock();
try {
   ... Sección crítica ...
} finally {
   cerrojo.unlock();
}
```

Variables de condición asociadas a cerrojos

- Una variable de condición o cola de eventos, permite dejar hilos bloqueados hasta que se cumpla una condición
- Cada variable de condición está asociada a un solo cerrojo, pero un cerrojo puede tener múltiples variables de condición
- Se puede despertar a los hilos de una cola (variable de condición) sin despertar a los que esperan en otra cola



Variables de condición asociadas a cerrojos

Implementación en java

```
Lock cerrojo;
Condition vc = cerrojo.newCondition();
...
cerrojo.lock();
try {
  while (! condición_necesaria)
    vc.await();
  ... Sección crítica ...
  vc.signal/signalAll();
} finally {
  cerrojo.unlock();
}
```

Variables de condición asociadas a cerrojos

- Cuando se despierta a un hilo vuelve a la cola asociada al cerrojo, y cuando consigue cerrarlo puede evaluar la condición de nuevo
- En java, las operaciones sobre una variable de condición son:
  - await() Bloquea al hilo hasta que otro lo despierte.
  - signal() Si hay hilos bloqueados en la variable de condición, despierta a uno de ellos.
  - signalAll() Si hay hilos bloqueados en la variable de condición, los despierta a todos.

Semáforos (Dijkstra, 1965)

- Inicialización (S = x)
- Definición de la operación P (o Acquire o Wait)
  - Operación atómica
  - Si S > 0, decrementa S
  - En otro caso, se bloquea
- Definición de la operación V (o Release o Signal)
  - Operación atómica
  - Si hay hilos bloqueados, desbloquea a uno
  - En otro caso, incrementa S

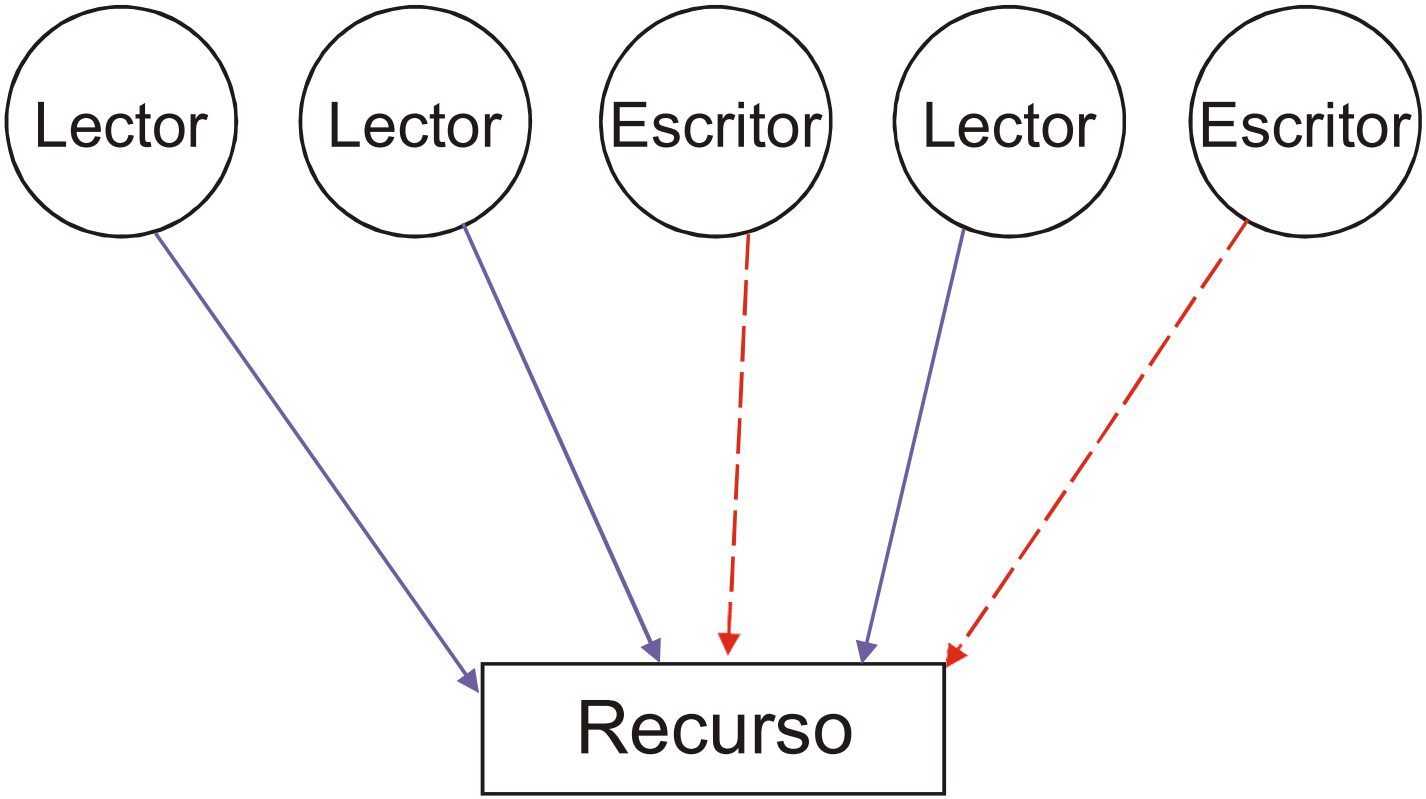
Monitores (Hoare)

- Herramienta de sincronización que permite a los hilos:
  - Ejecutar las operaciones sobre el monitor en exclusión mutua.
  - Esperar de manera inactiva a que se cumpla una condición.
  - Avisar a los demás hilos cuando se ha producido una condición.
- En java, cualquier clase puede ser usada como monitor:
  - Declarando **synchronized** todos los métodos públicos.
  - Esperar a que se cumpla una condición B:
    - while (! B) wait();
  - Informar a los demás hilos de que su condición puede haber cambiado
    - notify();
    - notifyAll();

Problema de los lectores-escritores

Restricciones en el acceso al recurso compartido:

- Unos hilos (lectores) pueden acceder concurrentemente
- Otros hilos (escritores) deben acceder en exclusión mutua frente a todos los demás hilos (lectores y escritores)

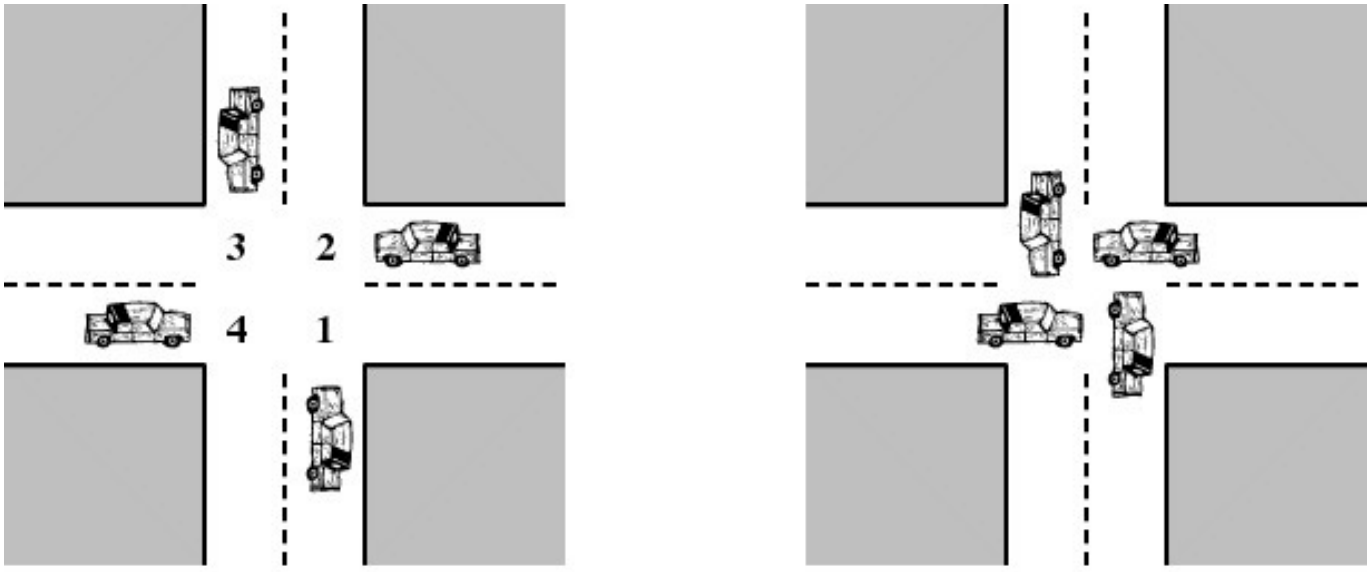


Cerrojo de lectores/escritores

- Un tipo especial de cerrojo con dos modos de acceso:
  - Modo "lector": varios hilos pueden cerrar el cerrojo concurrentemente
  - Modo "escritor": un único hilo puede cerrar el cerrojo en exclusión mutua
- Mejor que un cerrojo tradicional si las lecturas son mucho más frecuentes que las escrituras

Interbloqueo

Bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o se comunican entre sí. Problema dependiente de la velocidad de ejecución de los procesos



(a) Posible interbloqueo

(b) Interbloqueo

(a) Deadlock possible

(b) Deadlock

Representación del interbloqueo.

Interbloqueo. Ejemplo

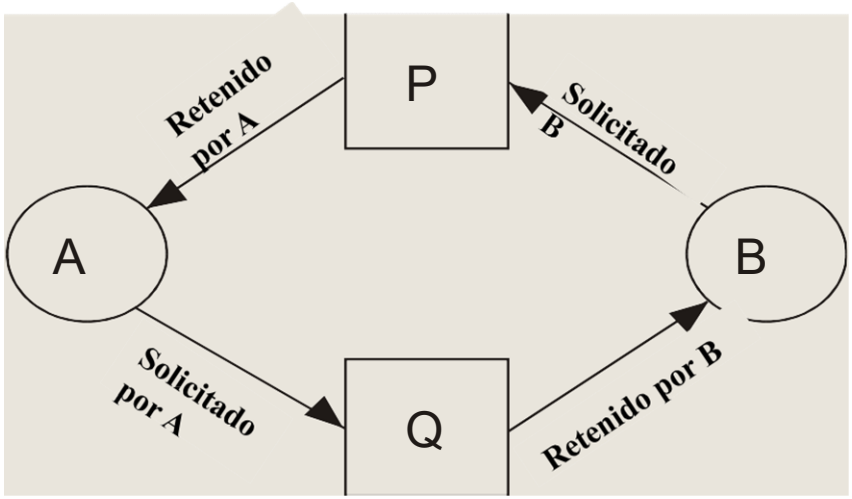
- A y B son dos hilos. P y Q son dos cerrojos

A

```
P.lock();
  ...
Q.lock();
  ...
P.unlock();
  ...
Q.unlock();
```

B

```
Q.lock();
  ...
P.lock();
  ...
Q.unlock();
  ...
P.unlock();
```



Condiciones de interbloqueo

Se deben dar estas 4 condiciones:

- Exclusión mutua:

  Los procesos acceden a los recursos en exclusión mutua

- No apropiación:

  Un recurso sólo puede ser liberado voluntariamente por el proceso que lo tiene asignado

- Retención y espera:

  Los procesos retienen los recursos que han conseguido hasta el momento y esperan para conseguir otros que están siendo usados

- Espera circular:

  Ha de existir una espera circular de dos o más procesos, cada uno de los cuales espera para captar un recurso que está utilizando el siguiente proceso del círculo

Apéndice

Aplicaciones multihilo y herramientas de concurrencia en Java

- Hilos
- Regiones criticas con synchronized
- Regiones criticas condicionales con synchronized
- Regiones criticas con cerrojos
- Variables de condición asociadas a cerrojos
- Semáforos
- Monitores
- Objetos atómicos
- Colas concurrentes
- Mapas concurrentes
- Modelo de memoria y concurrencia

Objetos atómicos en java

- Objetos cuyas operaciones son todas atómicas
- Disponibles en el paquete **java.util.concurrent.atomic**
- Hay que valorar si interesa, puesto que tienen algunas limitaciones frente a las variables estándar
- Ejemplos
  - AtomicInteger
    - AtomicInteger (int);
    - addAndGet (int); getAndAdd (int);
    - decrementAndGet(); incrementAndGet();
    - get(); getAndDecrement(); getAndIncrement();
  - AtomicBoolean
    - AtomicBoolean (boolean);
    - getAndSet (boolean);
    - compareAndSet(esperado, nuevo);

Objetos atómicos en java

Ejemplos de uso de AtomicInteger:

```
int n;
AtomicInteger numAtomico;

numAtomico = new AtomicInteger (5); n  =
numAtomico.getAndIncrement(); n =    // n = 5
numAtomico.get();                    // n = 6

numAtomico = new AtomicInteger (10);
n = numAtomico.get();                // n = 10
n = numAtomico.addAndGet(20);        // n= 30
```

Colas concurrentes

- BlockingQueue: interfaz de cola concurrente
  - ArrayBlockingQueue: implementación de cola FIFO concurrente
  - Operaciones:
    - Insertar un elemento:
      - *offer*. No bloqueante
      - *put*. Bloqueante
    - Obtener un elemento:
      - *poll*. No bloqueante
      - *take*. Bloqueante
- Otras BlockingQueue:
  - LinkedBlockingQueue
  - PriorityBlockingQueue
  - …

Mapas concurrentes

ConcurrentMap: interfaz de mapa concurrente

ConcurrentHashMap: implementación de mapa concurrente

```
ConcurrentMap mapa;
mapa = new ConcurrentHashMap<String,AtomicInteger>();
```

Resumen de clase y métodos

class Thread	Thread() Thread(Runnable target) static Thread currentThread() void interrupt() void join() throws InterruptedException void run() static void sleep(long millis) throws InterruptedException void start()
interface Runnable	void run()
interface Lock	void lock() Condition newCondition() void unlock()
class ReentrantLock	implements Lock ReentrantLock()
interface Condition	void await() throws InterruptedException void signal() void signalAll()
interface ReadWriteLock	Lock readLock() Lock writeLock()
class ReentrantReadWriteLock	implements ReadWriteLock ReentrantReadWriteLock()
class Semaphore	Semaphore(int permits) void acquire() throws InterruptedException void acquire(int permits) throws InterruptedException void release() void release(int permits)
interface BlockingQueue<E>	void put(E e) throws InterruptedException E take() throws InterruptedException
class ArrayBlockingQueue<E>	implements BlockingQueue<E> ArrayBlockingQueue(int capacity) void clear() int size()
class AtomicInteger	AtomicInteger(int initialValue) int addAndGet(int delta) ; int getAndAdd(int delta) int getAndIncrement() ; int getAndDecrement() int incrementAndGet() ; int decrementAndGet() int get() ; void set(int newValue) int getAndSet(int newValue) boolean compareAndSet(int expect, int update)