Primer Parcial

Tema 1 - Introducción a la Programación Concurrente

Programa concurrente: colección de actividades (hilos) que pueden ejecutarse en paralelo y cooperan para llevar a cabo una tarea común.

Ventajas

- **Eficiencia:** explota mejor los recursos
- **Escalabilidad**: puede extenderse
- **Gestión de comunicaciones:** explota la red
- **Flexibilidad:** resulta más fácil adaptar el programa
- **Menor hueco semántico**

Inconvenientes

- **Programación delicada**
- **Depuración compleja**

Creación de hilos en Java

Clase con nombre

Implementando Runnable

```
public class H implements Runnable
 public void run() {
  System.out.println("ejecuta hilo");}
}
Thread t= new Thread(new H());
t.start();
```

Extendiendo Thread

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

Clase anónima

Implementando Runnable

```
new Thread(new Runnable() {
 public void run() {
  System.out.println("ejecuta hilo");}
}).start();
```

Extendiendo Thread

```
new Thread() {
 public void run() {
  System.out.println("ejecuta hilo");}
}.start();
```

(si la clase extiende de otra, solamente se puede implementar Runnable, no se soporta la herencia múltiple)

Métodos Útiles

Identificación de hilos

Al crear un hilo se puede asociar un nombre:

setName -> **t.setName("thread" + i);** en cualquier momento se le puede asociar un nombre al hilo con este método

getName -> **t.getName();** identificador accesible sobre cualquier objeto Thread

(ejemplo): **Thread.currentThread().getName();**

Pausar la ejecución

Thread.sleep(long millis) -> causa la suspension de la ejecución del hilopor el tiempo indicado, lanza la excepción InterruptedException cuando el hilo suspendido es interrumpido por otro hilo

(ejemplo):

```
try {
      Thread.sleep(4000);
   } catch (InterruptedException e) {
     System.out.println("Caught InterruptedException: " + e.getMessage() );
   }
```

Interrumpir el hilo

Thread.Interrupt() -> operacion que reactiva un hilo que estaba suspendido, el hilo interrumpido recibe InterruptedException

Esperar a la terminación de un hilo

t.join() -> el hilo actual espera a que el hilo t termine

Thread.join(long millis) -> se puede especificar un tiempo máximo de espera

Se puede interrumpir al hilo que espera con el método Thread.Interrupt();

Otros Métodos

Thread.currentThread() -> devuelve referencia al objeto que se esta ejecutando en ese momento

Thread.isAlive() -> devuelve TRUE si el hilo se ha iniciado y no ha terminado, FALSE si es el caso contrario

Thread.yield() -> abandona el procesador

Tema 2 - Sincronización de tareas

Mecanismos de comunicación

Memoria compartida

Los hilos comparten espacio de memoria (variables u objetos compartidos)

Requiere mecanismos de sincronización para coordinar las tareas

Posibles problemas:

Condiciones de carrera: modificación inconsistente de la memoria compartida

- Pueden aparecer errores cuando múltiples hilos acceden a datos compartidos
- Pueden aparecer errores de vistas inconsistentes de la memoria compartida

(con las sincronización se previenen estos problemas)

Intercambio de mensajes

Emisor/receptor potencialmente en espacios de memoria disjuntos

Tipos de sincronización

Exclusión mutua

Solamente un hilo puede ejecutar la sección en cada momento, evita interferencias entre hilos

Sección critica

Fragmento que accede a variables u objetos compartidos por más de un hilo

Locks

Es un objeto con dos estados (abierto y cerrado) y dos operaciones (abrir y cerrar)

El lock al ser creado, se encontrará originalmente abierto. El uso es el siguiente: se cierra cuando se va a acceder a la sección critica, y después de haber accedido, al salir, se vuelve a abrir el lock para que otro hilo pueda entrar a esta.

Operaciones:

Cerrar lock

- Si esta abierto se cierra
- Si esta cerrado por otro hilo se suspende
- Si esta cerrado por si mismo no tiene efecto

Abrir lock

- Si esta abierto no tiene efecto
- Si esta cerrado por otro hilo no tiene efecto
- Si esta cerrado por si mismo se abre el lock

Al usar locks, la sección critica se convierte en una acción atómica (Atomicidad: únicamente un hilo puede ejecutar el código protegido en un momento dado), esto garantiza un uso libre de condiciones de carrera, además todo hilo siempre tomara el valor mas reciente de cada variable.

Implementación de Locks

Se añade la etiqueta synchronized a todo método que modifique un atributo que luego va a tener que leer otro hilo o lea un atributo actualizado por otro hilo.

(ejemplo)

```
public synchronized void add(long x) {
  count += x;
}
```

- **Métodos sincronizados**: utilizando synchronized en la declaración del metodo
- **Sentencias sincronizadas:** indicando synchronized en el objeto, esto permite usar mas de un lock dentro de un metodo

Sincronización condicional

Un hilo se suspende hasta que cumpla determinada condición, la condición depende del valor de alguna variable compartida. Otros hilos al modificar esas variables harán que se cumplan las condiciones, reactivando así a los diferentes hilos suspendidos

Tema 3 - Primitivas de sincronización

Monitor

Un monitor es una clase para definir objetos que podemos compartir de forma segura entre distintos hilos

Sus métodos se ejecutan en exclusión mutua

Dispone de una cola de entrada donde esperan aquellos hilos que se desean utilizar el monitor cuando lo está utilizando otro hilo

Resuelve la sincronización

Podemos definir colas de espera dentro del monitor

Si un hilo que ejecuta código necesita esperar hasta que se cumpla una determinada condición:

- **C.wait() ->** deja libre el monitor y espera sobre la cola de la condición de c (para esperar)

Cuando otro hilo modifica el estado del monitor:

- **C.notify() ->** reactiva un hilo que espera en la cola de la condición c (para notificar a quien espera)

Variantes de monitor

Problema:
Un hilo activo (W) en el monitor ejecuta c.wait() y pasa a esperar sobre c, el monitor queda libre. Otro hilo (N) que estaba esperando pasa a ser el activo, si este hilo ejecuta c.notify() reactivaría a W pero solo uno (o W o N) puede continuar activo en el monitor.

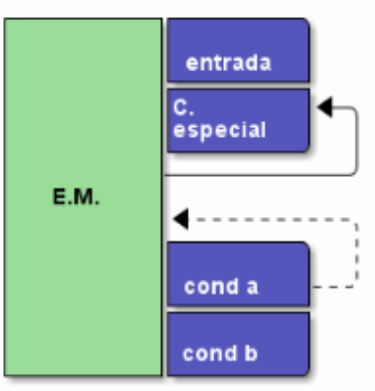
Soluciones:

- El hilo N espera en una cola especial (modelo Hoare)
- El hilo W espera en la entrada (modelo Lampson-Redell)

Monitor tipo Hoare

Existe una cola extra, la cola especial la cual es prioritaria a la cola de entrada y sirve para esperar a que el monitor quede libre.

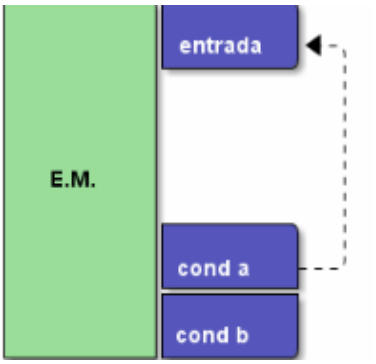
Cuando N ejecuta c.notify() y reactiva W, N pasaría a la cola especial mientras que W quedaría activo en el monitor, lo que nos proporcionaría exclusión mutua.



Monitor tipo Lampson-Redell

Cuando N ejecute notify() y reactive a W, W pasara a la cola de entrada mientras que será N el que queda activo en el monitor.

(cuando consiga entrar el estado puede haber cambiado de nuevo, tendríamos que reevaluar la condición)



Java – Monitor

Todo objeto posee de forma implícita:

- **Un Lock**
  Al etiquetar un método con synchronized se garantiza la exclusión mutua.

- **Una cola de espera con primitivas**
  wait() -> espera sobre la cola de espera
  notify() -> reactiva a uno de los hilos que esperan en dicha cola
  notifyAll() -> reactiva todos los hilos que están esperando

Como definir un monitor

Una clase que defina objetos a compartir debe:

- Definir todos sus métodos como privados
- Sincronizar los no privados (usando synchronized)
- Utilizar wait(), notify(), notifyAll() dentro de los métodos sincronizados

Los hilos que esperan por condiciones lógicas distintas deben esperar en colas de espera distintas

(ejemplo) en un escenario productor consumidor, los productores que encuentran buffer lleno esperarían en noLleno, y los consumidores que esperan porque el buffer está vacío esperarían en noVacio.

Interbloqueos

Situaciones provocadas cuando se produce un bloqueo en el cual no puede continuar la ejecución ningún hilo.

(ejemplo de interbloqueo)

```
class BCell {
  int value;
  public synchronized void getValue() {
    return value;
  }
  public synchronized void setValue(int i) {
     value=i;
  }
  public synchronized void swap(BCell x) {
    int temp= getValue();
      setValue(x.getValue());
      x.setValue(temp);
 }
}
```

Explicación

El primer hilo invoca p.swap(q), obtiene acceso al monitor “p”, e inicia la ejecución de p.swap, El segundo hilo invoca q.swap(p), obtiene acceso al monitor “q”, e inicia la ejecución de q.swap.

Dentro de p.swap, el primer hilo invoca q.getValue(), pero debe esperar porque el monitor q no está libre. Dentro de q.swap, el segundo hilo invoca p.getValue(), pero debe esperar porque el monitor p no está libre.

Conclusión

Los dos hilos se están esperando mutuamente y ninguno le puede dar el valor que necesita al otro, por lo que se produciría un interbloqueo ya que la situación no puede evolucionar.

Tema 4 - Interbloqueos

Conceptos

Recurso: cualquier elemento físico o lógico que solicita un hilo, un recurso puede tener varias instancias. Un hilo A espera a un hilo B cuando solicita un recurso usado por B y este no es compartible
Los hilos usan recursos con el protocolo petición – uso - liberación

Interbloqueos: conjunto de hilos que no pueden evolucionar porque se esperan mutuamente

Condiciones de Coffman

Si todas se cumplen simultáneamente, hay riesgo de interbloqueo. Que se cumplan todas no quiere decir que haya interbloqueo, pero si hay interbloqueo, se cumplen todas.

Exclusión mutua

Mientras un recurso está asignado a un hilo, otros no pueden usarlo

Retencion y espera

Los recursos se solicitan a medida que se necesitan, de forma que podemos tener un recurso asignado y solicitar otro no disponible (en espera)

No expulsión

Un recurso asignado solo o puede liberar su dueño

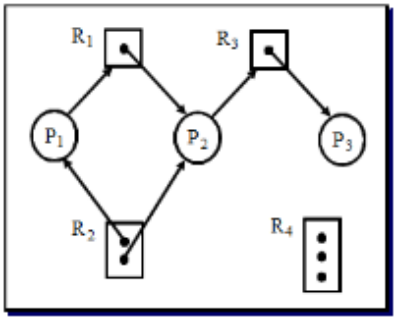
Espera circular

En el grupo de hilos interbloqueados, cada uno esta esperando un recurso que tiene otro del hilo, así hasta cerrar el circulo

Representación Gráfica: GAR

Es una representación gráfica del estado del sistema, teniendo como elementos:

- **Un círculo** representa al hilo
- **Un rectángulo** representa el recurso, y cada punto interno que tiene es una instancia del recurso
- **Arista de asignación:** un recurso asignado es una flecha desde una instancia concreta al hilo que la utiliza
- **Arista de petición:** una solicitud no resuelta es una flecha que va desde el hilo al recurso

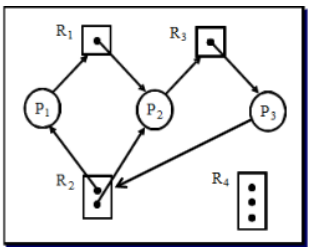


(en la imagen)

P1, P2 y P3 son hilos
R1, R2, R3, R4 son recursos, con 1,2,1 y 3 instancias respectivamente
La instancia de R1 esta asignada a P2, la de R3 esta asignada a P3, y las de R2 están asignadas a P1 y a P2
P1 esta esperando una instancia de R1 y P2 una instancia de R3

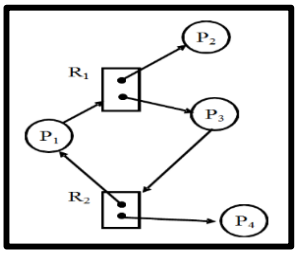
Ciclo dirigido

Si todos los recursos que participan en el ciclo tienen una instancia y el ciclo dirigido, tenemos un interbloqueo



Secuencia Segura

Un orden en el cual pueden terminar todos los hilos, con esto nos aseguraríamos de que no hay interbloqueo



Solución (de mejor a peor)

Prevención

Diseñar un sistema que rompa alguna condición de Coffman, con esto nos aseguraríamos de que no es posible el interbloqueo

Evitación

Usando un GAR, el sistema considera cada solicitud y si es seguro atenderla en ese momento, si una solicitud crea un riesgo de interbloqueo, la deniega

Detección y recuperación

Monitoriza periódicamente el sistema, si se ha llegado a interbloqueo, se abortan algunas actividades involucradas

Ignorar el problema

No resuelve nada, cómodo y frecuente

Prevención

La prevención implica romper alguna condición de Coffman

Exclusión Mutua

En muchos casos la mayor parte de los recursos deben usarse en exclusión mutua

Retención y espera

La forma normal de trabajar es solicitar recursos a medida que los necesitamos:

- **Solución 1:** Pedir en el inicio todo lo que podamos necesitar
- **Solución 2:** Solicitar recursos de forma no bloqueante (si el recurso esta en uso, recibe un valor que indica esta situación)

No expulsión

Se permite que un hilo expropie recursos de otro, el hilo expropiado tendría que volver a solicitar recursos, podríamos tener varios hilos que no avancen

Espera circular

Se establece un orden total entre los recursos y se les obliga a solicitarlos en orden, la más fácil de romper

Tema 5 - Otras Herramientas de Sincronización

Biblioteca java.util.concurrent

Ofrece instrucciones de alto nivel que garantizan:

- **Productividad:** facilita el desarrollo y mantenimiento de aplicaciones concurrentes de calidad
- **Prestaciones:** más eficiente y escalable
- **Fiabilidad:** Comprobaciones extensivas contra interbloqueos, condiciones de carrera e inanición

Locks

Permite especificar si se requiere una gestión equitativa de la cola de espera mantenida por el lock. Ofrece un método trylock() que no suspende al invocador si el lock ya ha sido cerrado por otro hilo.

ReentrantLock

Dentro de la sección de código protegida por el lock se podrá volver a utilizar el mismo lock sin que haya problemas de bloqueo

¿Qué resuelve?

Resuelve las limitaciones de synchronized:

- **tryLock() con timeout -> permite especificar un plazo máximo de espera para obtener el lock**
- **newCondition() -> permite definir distintas variables condición**
- **lock() -> cerrar el lock**
- **unlock() -> abrir el lock**
- **thread.interrupt() -> interrumpir las esperas**

Inconvenientes

El lock no se abre por defecto al terminar un bloque de código, se tiene que hacer manualmente, hay que comprobar las excepciones que se puedan generar dentro de una sección critica protegida con un ReentrantLock

Estructura

```
Lock x = new ReentrantLock();
x.lock();
try {
.. //Sección crítica  (donde se actualiza el estado del objeto)
} finally {
   x.unlock();
}
```

Protocolo de entrada

Protocolo de salida

Variables condición

La interfaz Condition permite declarar cualquier número de variables condición en un lock (múltiples variables, en muchos casos, permiten un código más legible y eficiente)

Métodos

- **await() ->** suspende a un hilo en la condición
- **signal() ->** notifica a uno de los hilos que estuviera esperándolo
- **signalAll() ->** notifica a todos los hilos suspendidos en la condición

Variables atómicas

Define clases que soportan el acceso concurrente seguro a variables simples
Tipos primitivos:

- **AtomicBoolean**
- **AtomicInteger**
- **AtomicLong:** permite actualizar de forma atómica un valor long

Es útil para implementar algoritmos concurrentes de forma eficiente, para la implementación de contadores y la generación de secuencias de números.

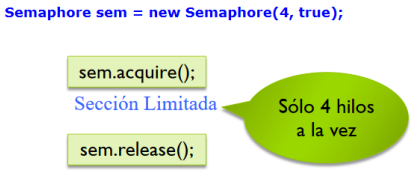
Semáforos

Permite la sincronización entre hilos, se usan los métodos:

- **acquire():** espera hasta disponer de un permiso y lo consume
- **release():** añade un permiso y probablemente libera a un hilo esperando tras acquire

El semáforo lleva asociado un contador que se inicializa en la creación, si este se inicializa a uno, se garantiza exclusión mutua

Si se inicializa a un valor positivo mayor que uno, limita el grado de concurrencia



Si se inicializan a cero, se garantiza cierto orden de ejecución entre dos o más hilos

Barreras

Permiten sincronizar a múltiples hilos de ejecución

Cyclic Barrier

Permite que varios hilos se esperen mutuamente en un punto, cada vez que un hilo llega a el punto, se usa el método await() para que este se suspenda, cuando el ultimo hilo invoca a await(), la barrera se abre por lo que todos los hilos se reactivan, es reutilizable (al abrirse y pasar los hilos, se restauran las condiciones iniciales)

(ejemplo)

```
class Solver {
 final int N;
 final float[][] data;
 final CyclicBarrier barrier;
 class Worker implements
Runnable {
   int myRow;
   Worker(int row) { myRow = row; }
   public void run() {
     while (!done()) {
       processRow(myRow);
       try { barrier.await();
      } catch (InterruptedException e)
          {return;}
       catch (BrokenBarrierException
e)
         {return;} } } }
```

```
public Solver(float[][] matrix) {
  data = matrix;
  N = matrix.length;
  barrier = new CyclicBarrier(N,
    new Runnable() {
      public void run() {
        mergeRows(...);
        }
  });
  for (int i = 0; i < N; ++i)
    new Thread(new
Worker(i)).start();
    waitUntilDone();
  }
}
```

CountDownLatch

Permite suspender a un grupo de hilos, a la espera de que suceda algún evento generado por otro hilo ajeno, se mantiene un contador de eventos.

Métodos

- **Await() -> los hilos se bloquean en la barrera mientras esté cerrada**
- **countDown() -> decrementa en 1 el valor del contador, si el contador pasa a 0 la barrera se abre liberando todos los hilos.**

Es una barrera de un solo uso, una vez el contador llega a cero, permanece así, para poder reutilizar se debería usar una CyclicBarrier

(ejemplo)

```
class Driver { // ...
  void main() throws
InterruptedException {
    CountDownLatch startSignal =
new CountDownLatch(1);
    CountDownLatch doneSignal =
new CountDownLatch(N);
    for (int i = 0; i < N; ++i)
      new Thread(new
Worker(startSignal,
doneSignal)).start();
    doSomethingElse();
    startSignal.countDown(); //para
qur todos los worker comiencen a
la vez
    doSomethingElse();
    doneSignal.await(); //donde el
driver espera a que todos los
worker finalicen su trabajo
  }
}
```

```
class Worker implements Runnable
{
  private final CountDownLatch
startSignal;
  private final CountDownLatch
doneSignal;
  Worker(CountDownLatch start,
CountDownLatch done) {
    startSignal = start;
    doneSignal = done;
  }
  public void run() {
    try { startSignal.await();
      doWork();
      doneSignal.countDown();
    } catch (InterruptedException ex)
{return;}
  }
  void doWork() { ... }
}
```

Interfaz Executor

Hilo único (SingleThreadExecutor)

Utiliza un solo hilo de trabajo que opera en una cola sin límites, garantiza que las tareas se ejecuten de forma secuencial y que no habrá más de una tardea activa en un momento dado

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
```

Un thread-pool

Permite mantener un conjunto de hilos ya creados, reciclándolos para que ejecuten nuevas tareas

```
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
```

(ejemplo completo)

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
executorService.execute(new Runnable() {
  public void run() {
    System.out.println("Asynchronous task");
  }
});
executorService.shutdown();
```

Detener el ExecutorService

shutdown()

Permite detener los hilos del Executor, no se detiene inmediatamente pero ya no aceptará nuevas tareas, cuando hayan acabado todos los hilos con sus tareas actuales se detendrá

shutdownNow()

Trata de detener todas las tareas en ejecución, las tareas presentadas pero no procesadas las ignora, intenta parar las tareas que se encuentran en ejecución y devuelve la lista de las tareas que no finalizaron.