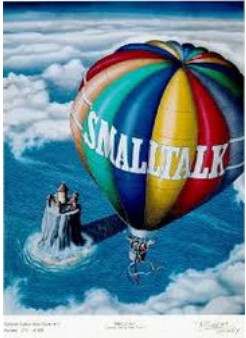




Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue



Programación Concurrente



*Instrumentos de la
concurrency*



Mecanismo de sincronización: monitor

- Un monitor es un mecanismo de abstracción de datos que se aplica en un ambiente concurrente
- Un monitor asegura exclusión mutua
- Un monitor es un objeto pasivo, que generalmente es un recurso compartido
- Un proceso que invoca un método sobre un monitor puede ignorar cómo el método es implementado. Todo lo que importa son los efectos visibles de la invocación
- El programador de un monitor puede ignorar cómo y dónde los métodos del monitor son usados.

Mecanismo de sincronización: monitor

- Dispone de los métodos:
wait(), notify(); notifyAll()
- Cada objeto tiene un *lock* y un *conjunto de espera* (CE)
- Cualquier objeto puede servir como *monitor*. Aunque cada lenguaje define sus propios detalles
- Cada CE mantiene los hilos bloqueados por un *wait* sobre el objeto
- Los CE interactúan con los locks, entonces *wait*, *notify* y *notifyAll* requieren sincronización.

Mecanismo de sincronización: monitor

wait(): acciones

- El hilo corriente es bloqueado / si el hilo fue interrumpido dispara una excepción “IntExc”
- El hilo es ubicado en el CE del objeto sobre el que es invocado
- El lock de sincronización del objeto
- se libera (SOLO ese lock)
- Si es *wait(msecs)* entonces transcurrido ese tiempo es liberado



Mecanismo de sincronización: monitor

notify(): acciones

- Si el CE no está vacío, un hilo H es elegido arbitrariamente y liberado.
- H debe volver a obtener el lock del objeto. H se bloquea hasta que el hilo que hizo el notify libere el lock.
- Si algún otro hilo obtiene el lock primero, continuará bloqueado.
- H continúa desde el punto de su wait.
- *notifyAll()* : todos los hilos que están en el CE son liberados

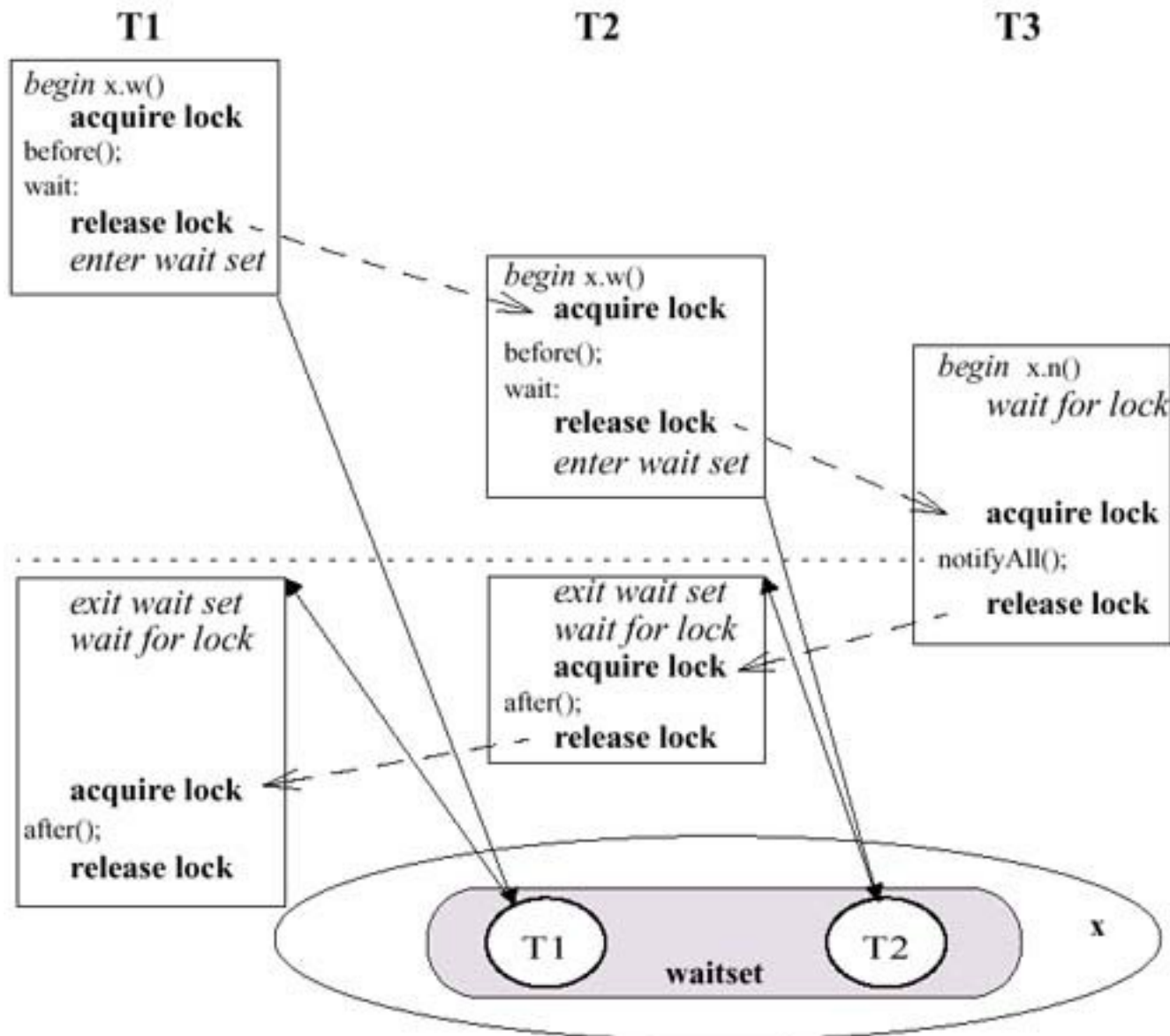
Mecanica de monitores

Consideremos el siguiente ejemplo inutil

```
class MonX {  
    synchronized void w() throws InterruptedException {  
        this.before();  
        this.wait();  
        this.after();  
    }  
  
    synchronized void n() {  
        notifyAll();  
    }  
  
    void before() {}  
    void after() {}  
}
```

Y consideremos 3 hilos que invocan los métodos del monitor MonX sobre una instancia compartida x, (x instancia de MonX)

Ejemplo tomado de: *“Concurrent Programming in Java” - Doug Lea*



Mecanismo de sincronización: monitor

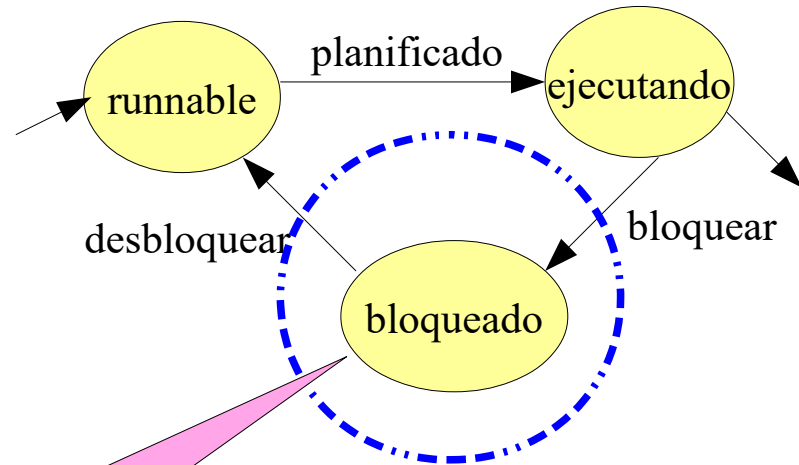
Esperas guardadas (dependen de una condición)

- En general las invocaciones de `wait()` se hacen dentro de un bucle `while`.
- Cuando una acción es reasumida, la tarea en espera NO SABE si la condición por la que estaba esperando es verdadera.
- Solo sabe que lo liberaron, y debe volver a verificar, para mantener la propiedad de seguridad.
- Es una buena práctica que este estilo se utilice aun cuando haya una única instancia que pueda esperar por la condición.

POO concurrente - Viveza

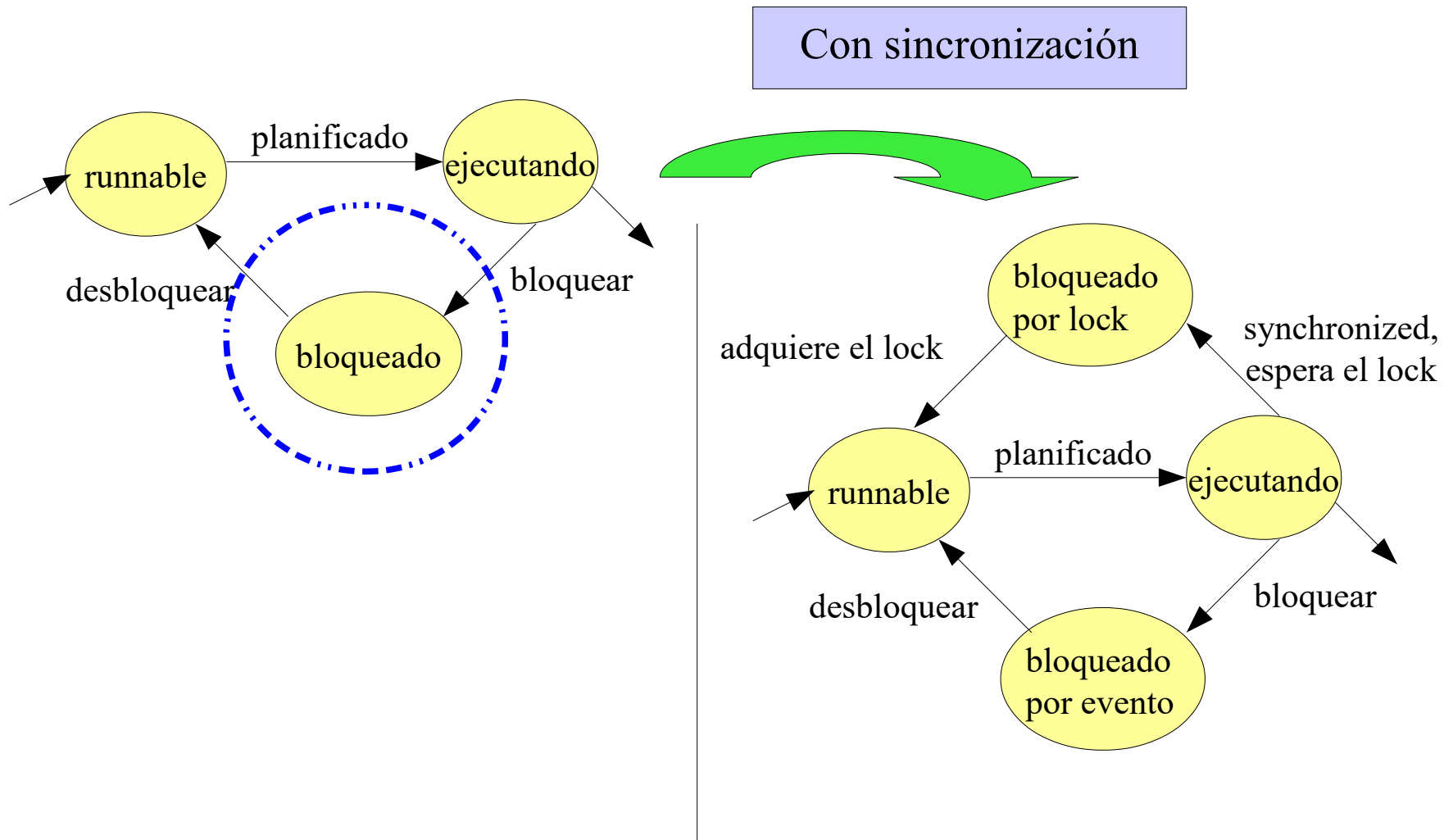
Pero ... una actividad puede fallar en progresar por alguna razón ...

- Bloqueo - cerrojo
- Espera - wait
- Entrada
- Contención de CPU
- Falla



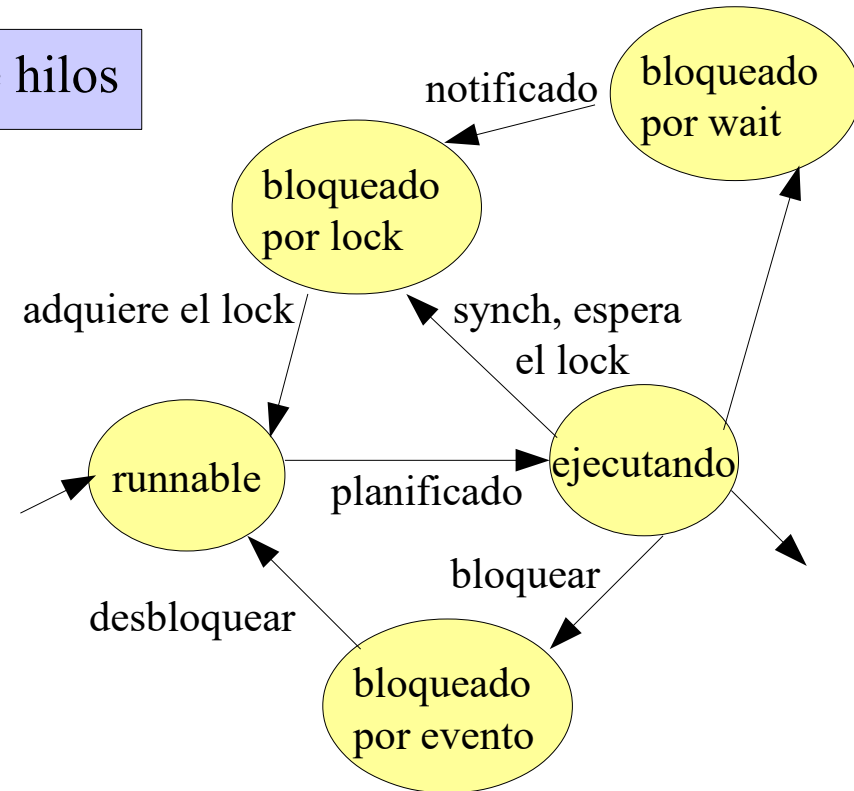
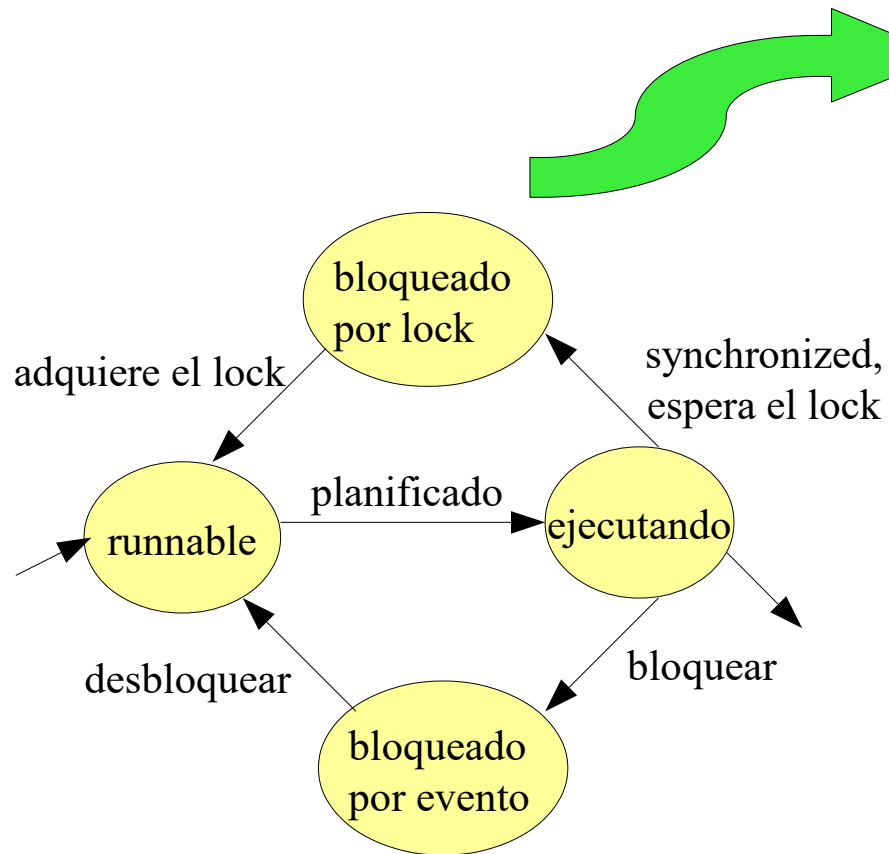
Representa
distintas situaciones
de bloqueo

Entonces ...



Entonces ...

Con sincronización e interacción entre hilos



Métodos de Object para sincronizar

- *public final void wait() throws InterruptedException*
 - Espera indefinida hasta que reciba una notificación.
- *public final void wait(long timeout) throws InterruptedException*
- *public final void wait(long timeout, int nanos)*
 - El thread que ejecuta el método se suspende hasta que recibe una notificación o luego de ese tiempo
- *public final void notify()*
 - Notifica al objeto un cambio de estado, esta notificación es transferida a solo uno de los threads que esperan
- *public final void notifyAll()*
 - Notifica a todos los threads que esperan (han ejecutado un wait) sobre el objeto.

Cómo utilizar el método *wait()*

- Utilizar dentro de un método *synchronized* y dentro de un ciclo indefinido que verifica la condición:

```
synchronized void hacerCondicion() {  
    while (!Condicion)  
        this.wait(); //condición cierta  
}
```

- Cuando se suspende el thread en el **wait**, **libera el lock** que poseía sobre el objeto.
- La suspensión del thread y la liberación del lock son atómicos (nada puede ocurrir entre ellos).
- La **activación del thread** y la toma del **lock del objeto** son también atómicos

Cómo usar el método *notify()*

- Utilizar dentro de un método *synchronized* :

```
synchronized void cambiaCondicion() {  
    ...//algo cambia y la condición se cumple  
    this.notifyAll() ;  
}
```

- Muchos thread pueden estar esperando sobre el objeto:
 - Con **notify()** solo un thread (no se sabe cual) es despertado.
 - Con **notifyAll()** todos los thread son despertados y cada uno decide si la notificación le afecta, o si no, vuelve a ejecutar el wait().
- El proceso suspendido debe esperar hasta que el procedimiento que invoca **notify()** o **notifyAll()** libere el lock del objeto.

Ejemplos de sincronización

- La variable *sucede* es la que define que espere un thread y que prosiga el otro

Thread 1

```
public synchronized void esperaEvento() {  
    //realiza una vez para c/evento que puede no ser esperado  
    while(!sucede) {  
        try { this.wait(); }  
        catch (InterruptedException e) {}  
    }  
    System.out.println("sucede ");  
}
```

Thread 2

```
public synchronized notificaEvento() {  
    sucede=true;  
    this.notify();  
}
```

Comunicación entre threads

- El thread A está esperando que el thread B le envíe un mensaje

Thread A

```
public synchronized void esperaMensaje() {  
    try { this.wait(); }  
    catch (InterruptedException e) {}  
}
```

Thread A

```
public void esperaMensaje() {  
    while (tieneMensaje == false) {  
        Thread.sleep(100);  
    }  
}
```

Thread B

```
public synchronized void poneMensaje() {  
    ...  
    this.notify();  
}
```

Thread B

```
public void poneMensaje() {  
    ...  
    tieneMensaje = true;  
}
```


Ejemplo

- En una rotisería



Chef



Rotisería



Comida



Cliente

Ejemplo



Chef

- nroPlato int
- rotiseria Rotiseria
- nombre String

Chef (Rotisería roti, String nomChef)
+run() : void



Comida

- nroOrden int
- Comida (int nro)
+toString() : String
+getComida() : ordenNro



Rotisería

- comida Comida
- chef Chef
- micliente Cliente

Rotiseria (String nombreChef)
+getComida() : Comida
+getChef() : Chef
+getClient(): Cliente
+setComida(Comida c): void



Ciente

- rotiseria Rotiseria
- nombre String

Cliente (Rotiseria roti, String nomb)
+toString() : String
+run() : void

Como vamos a sincronizar?



Thread A

```
synchronized (this) {  
    while comida != null{  
        //hasta que el cliente  
        // se lleven la comida  
        this.wait();  
    }  
}  
....  
synchronized (cliente) {  
    // Creo mas comida  
    // Le indico a la rotisería  
    cliente.notify()  
}
```



Thread B

```
synchronized (this) {  
    while comida == null{  
        this.wait();  
    }  
}  
'''  
synchronized (chef) {  
    vacio comida = null;  
    chef.notify()  
}
```

POO concurrente - Seguridad

- Cada método público en cada clase debería llevar un objeto de un estado consistente a otro.
- Objetos seguros pueden ocasionalmente estar en estado de inconsistencia en medio de los métodos, pero no intentan iniciar nuevas acciones en estados inconsistentes

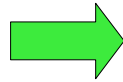
Cada objeto se diseña para ejecutar acciones solo cuando está habilitado lógicamente para ello

Todos los mecanismos son implementados de forma apropiada

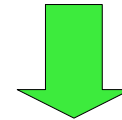
No hay errores por inconsistencia de objetos

POO concurrente - Seguridad

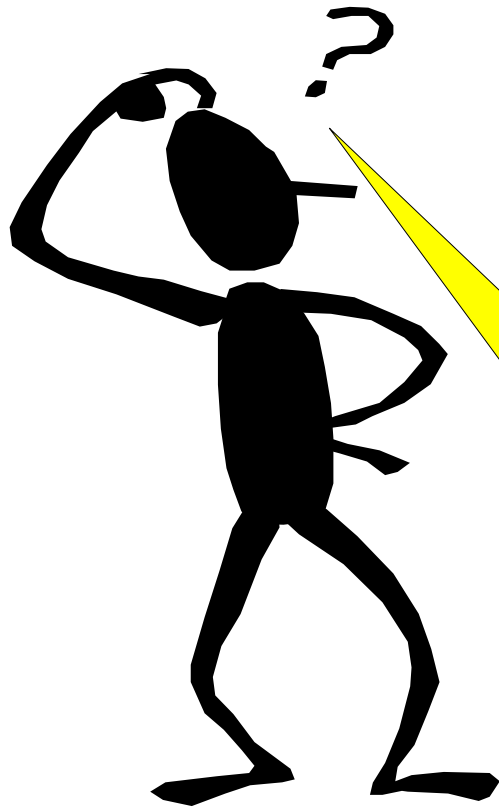
Asegurar consistencia



Emplear técnicas de exclusión




Garantizar la atomicidad de acciones públicas



Cada acción se ejecuta hasta que se completa sin interferencia de otras

Un balance de cuenta bancaria es incorrecto después de un intento de retirar dinero en medio de una transferencia automática

POO concurrente - Seguridad

- Inconsistencias  **condiciones de carrera**
- Conflictos de lectura/escritura: *un hilo lee un valor de una variable mientras otro hilo escribe en ella. El valor visto por el hilo que lee es difícil de predecir – depende de que hilo ganó la “carrera” para acceder a la variable primero*
- Conflictos de escritura/escritura: *dos hilos tratan de escribir la misma variable. El valor visto en la próxima lectura es difícil de predecir*

POO concurrente - Viveza

- Sistemas *vivos*
 - cada actividad eventualmente progresa hacia su finalización.
 - cada método invocado eventualmente se ejecuta
 - Pero ... una actividad puede fallar en progresar por alguna razón

POO concurrente - Viveza

- Problema serio: falta de progreso permanente
 - Deadlock: dependencias circulares
 - Livelock: una acción falla continuamente
 - Starvation: un hilo espera por siempre, la maquina virtual falla siempre en asignarle tiempo de CPU
 - Falta de recursos: un grupo de hilos tienen todos los recursos, un hilo necesita recursos adicionales pero no puede obtenerlos
 - Otros (investigar)

Exclusión

- En un sistema seguro cada objeto se protege de violaciones de integridad.
- Las técnicas de exclusión preservan los invariantes de los objetos y evitan efectos indeseados
- Las técnicas de programación logran la exclusión previniendo que múltiples hilos modifiquen o actúen de forma concurrente sobre la representación de los objetos.

Exclusión mutua: mantener estados consistentes de los objetos evitando interferencias NO deseadas entre actividades concurrentes

Exclusión

- Estrategias básicas:

- Eliminar la necesidad de control de exclusión asegurando que los métodos nunca modifican la representación de un objeto



inmutabilidad

- Asegurar dinámicamente que solamente 1 hilo por vez puede acceder el estado del objeto usando cerrojos y construcciones relacionadas



sincronización

- Asegurando estructuralmente que solamente 1 hilo por vez, puede utilizar un objeto dado ocultándolo o restringiendo el acceso a él.



confinamiento

Exclusion - inmutabilidad

- Si un objeto no puede cambiar su estado, nunca puede encontrar conflictos o inconsistencias cuando múltiples actividades intentan cambiar su estado.
- Los objetos inmutables mas simples no tienen campos internos, sus métodos son sin estado (*Stateless*)
- *Aplicaciones*
 - TDA: Integer, Date, Fraction, etc. - Instancias de estas clases nunca alteran los valores de sus atributos. Proveen métodos para crear objetos que representan nuevos valores
 - Contenedores de valores: es conveniente establecer un estado consistente una vez y mantenerlo por siempre.
 - Representaciones de estado compartidas: en general se trata de clases de utilidad.

Preguntas

El planificador de threads (scheduler)

- ¿Quién decide si un thread pasa de runnable a running?
- ¿Por qué se puede bloquear un thread?

Cuando intenta acceder a un objeto bloqueado, o cuando se queda durmiendo, o...

- ¿El programador puede influenciar en algo?

Si

- ¿Qué objetos tienen un lock?

Todos

- ¿Qué objetos se pueden sincronizar (synchronized) ?
- ¿Qué clase tiene los métodos wait(), notify() y notifyAll() ?
- ¿Sobre que objeto se los puede invocar?

Objeto thread propietario del lock