EXAMEN DE RECUPERACIÓN – PRIMER BLOQUE **Concurrencia y Sistemas Distribuidos**
Unidades Didácticas 1 a 6 - Prácticas 1 y 2 **23 de Junio de 2015**

Este examen tiene una duración total de 2 horas.

Este examen tiene una puntuación máxima de **10 puntos**, que equivalen a **4** puntos de la nota final de la asignatura. Indique, para cada una de las siguientes **60 afirmaciones**, si éstas son verdaderas (**V**) o falsas (**F**).

Cada respuesta vale: correcta= 1/6, errónea= -1/6, vacía=0.

Importante: Los **primeros 3 errores** **no penalizarán**, de modo que tendrán una valoración equivalente a la de una respuesta vacía. A partir del 4º error (inclusive), sí se aplicará el decremento por respuesta errónea.

Sobre el concepto de monitor y sus variantes:

1.	Un monitor es un mecanismo de sincronización de alto nivel integrado en algunos lenguajes de programación concurrentes.	V
2.	Un monitor que siga el modelo de Hoare suspende al hilo que invoca a *notify*(), quedándose dicho hilo suspendido en una cola especial.	V
3.	Un monitor que siga el modelo de Lampson/Redell jamás suspende a un hilo en la invocación a *notify*().	V
4.	Un monitor que siga el modelo de Brinch-Hansen requiere que toda invocación a *notify*() sea la última sentencia en los métodos del monitor en los que aparezca tal invocación.	V

Para construir monitores en Java...:

5.	... utilizaremos los métodos *wait*(), *notify*() y *notifyAll*() (heredados de la clase Object) para implantar la sincronización condicional.	V
6.	... debemos añadir el calificador "synchronized" a cada uno de los métodos públicos de la clase que vaya a utilizarse como un monitor.	V
7.	Una desventaja de los monitores estándar de Java la tenemos en que no es posible limitar el tiempo máximo que los hilos deben esperar para entrar a los monitores.	V

Sobre las herramientas de la biblioteca *java.util.concurrent*....

8.	Un objeto "c" de la clase *CountDownLatch* se puede utilizar para garantizar exclusión mutua, inicializándolo a 1 en su constructor, y protegiendo la sección crítica entre un *c.await()* a su entrada y un *c.countDown()* a su salida. *JUSTIFICACIÓN: Con esta implementación, todos los hilos que lleguen a la sección crítica se quedarán bloqueados, pues el código para abrir la barrera (el c.countDown) está justo a la salida de la sección crítica, de modo que ningún hilo podrá invocarla.*	F
9.	El método constructor de *CyclicBarrier* recibe como argumento un objeto de tipo "Runnable". Se ejecutará el método *run*() de este objeto cuando todos los threads que comparten la barrera hayan ejecutado el método *await*() de la barrera.	V
10.	Para utilizar un objeto *Condition* en un "monitor" protegido con *ReentrantLock* en Java se deben etiquetar los métodos donde se use con el calificador "synchronized". *JUSTIFICACIÓN: Si se etiquetan con synchronized, entonces se está aplicando un doble cerrojo (el del ReentrantLock y el cerrojo implícito de Java), de modo que cuando se abra el lock al realizar c.await(), se abrirá solamente el lock del ReentrantLock pero no el del synchronized, por lo que los hilos a la espera del lock no podrán progresar.*	F
11.	Los cerrojos de Java, presentes en la biblioteca "java.util.concurrent.locks", constituyen un mecanismo de sincronización útil para implementaciones que necesiten romper la condición de Coffman de retención y espera.	V
12.	Cada cerrojo de tipo *ReentrantLock* puede crear una única variable condición, de forma similar a como sucede con la única variable condición presente de forma implícita en los monitores estándar de Java. JUSTIFICACIÓN: Se pueden crear tantas variables condición como se requiera. Se crean utilizando el método newCondition() de ReentrantLock.	F
13.	Un objeto "c" de la clase *CountDownLatch* se puede utilizar para que M hilos de clase B esperen a que otro hilo A les avise. Para ello, inicializamos "c" a 1, los hilos B usarán *c.await*() y el hilo A llamará a *c.countDown*() una sola vez.	V

14.	Para que un hilo A espere hasta que otros N hilos de una misma clase (H1..Hn) hayan ejecutado una sentencia B dentro de su código se puede utilizar un *Semaphore* S inicializado a 0; A invoca *S.release()*, mientras que H1..Hn invocan *S.acquire()* tras la sentencia B. *JUSTIFICACIÓN: Con esta implementación, A no espera a nadie, y n-1 hilos H quedan bloqueados al realizar S.acquire().*	F

Sea el conjunto de tareas en un sistema de tiempo real descrito por la siguiente tabla:

Tarea	Periodo (T)	Cómputo (C)	Plazo (D)	Prioridad	Usa Si(t)
A	20	5	8	1	S1(2), S2(3)
B	15	3	12	2	S2(2), S3 (1)
C	10	4	15	3	S1(3), S3(2)

donde prioridad A>B>C; semáforos S1,S2,S3, y utilizando el protocolo de techo de prioridad inmediata, así como el algoritmo de planificación por prioridades fijas expulsivas.
Como se ve en la tabla, la tarea A utiliza S1 y S2 para secciones críticas de longitud 2 y 3 respectivamente. La tarea B utiliza S2 y S3 (longitudes 2 y 1), mientras que la tarea C utiliza S1 y S3 (longitudes 3 y 2, respectivamente).

15.	Siempre que C está en ejecución y llega B al sistema, B expulsa a C. *JUSTIFICACIÓN: La tarea C utiliza el semáforo S1 (cuyo techo es igual a la prioridad de A), así como el semáforo S3 (cuyo techo es igual a la prioridad de B). Al utilizar el protocolo de techo de prioridad inmediata, si C ha cerrado el semáforo S1 o bien el semáforo S3 antes de que B llegue al sistema, entonces habrá adquirido la prioridad asociada al techo del semáforo, es decir, prioridad(A) o prioridad(B). En ambos casos, si B llega entonces al sistema, no expulsaría a la tarea C, pues ésta tendrá una prioridad mayor o igual a la de B.*	F
16.	El factor de bloqueo de B es 2. *JUSTIFICACIÓN: Las tareas menos prioritarias que B es solamente C. Esta tarea utiliza el semáforo S1, con ceiling(S1)=prioridad(A), y el semáforo S3, con celing(S3)=prioridad(B). Por tanto, debemos considerar ambos semáforos, pues tienen techos mayores o iguales a la prioridad de B. Si vemos las longitudes de las secciones críticas que usa C con dichos semáforos, tenemos que $Usa_C(S1)=3$ y $Usa_C(S3)=2$, por lo que el máximo de ellas es 3. Por tanto, el factor de bloqueo de B es 3 ($B_B=3$).*	F
17.	El techo de prioridad de S1 y S3 es el mismo. *JUSTIFICACIÓN: Ceiling(S1)=prioridad(A) y Ceiling(S3)=prioridad(B).*	F
18.	El tiempo de respuesta de A es 8, el tiempo de respuesta de B es 11 y el tiempo de respuesta de C es 12. *JUSTIFICACIÓN: Debemos calcular primero los factores de bloqueo.* *B_A= 3 (pues $Usa_C(S1)=3$, y dicho valor es el máximo)* *BB=3 (calculado en preg. 16)* *BC=0 (no hay tareas menos prioritarias).* *Y calculando los tiempos de respuesta (considerando el factor de bloqueo):* *RA= 5 + 3 = 8 <=8* *RB= 11 W_B^0=3+5+3=11 <=12; W_B^1=3 + 3 + [11/20]*5=3 + 3+ 1*5=11 <=12* *R_C= 12 <=15 (D_C) W_C^0=4+3+5=12 <=15; W_C^1=4+[12/15]*3+[12/20]*5=12*	V
19.	Como queda garantizado el cumplimiento del plazo de todas las tareas, el sistema es planificable.	V
20.	Si las tareas fueran independientes y, por tanto, no utilizaran semáforos (pues no tendrían secciones críticas), el sistema sería planificable. *JUSTIFICACIÓN: Si las tareas fueran independientes, tendríamos:* *R_A= 5 <= 8 (D_A)* *R_B= 8 <=12 (D_B) W_B^0=3+5=8 <=12; W_B^1=3 + [8/20]*5=3 + 1*5=8 <=12* *R_C= 12 <=15 (D_C) W_C^0=4+3+5=12 <=15; W_C^1=4+[12/15]*3+[12/20]*5=12* *El sistema es planificable.*	V

Sobre la programación concurrente:

21.	Una de las desventajas de la programación concurrente es que dicha programación es más compleja porque requiere de técnicas para evitar las condiciones de carrera.	V
22.	Un proceso con un hilo de ejecución, ejecutado en un ordenador cuyo procesador tiene cuatro núcleos, es un ejemplo de aplicación concurrente. *JUSTIFICACIÓN: Solamente hay un hilo en ejecución (i.e. una única tarea), por tanto, no puede existir concurrencia entre varias tareas.*	F
23.	Una aplicación compuesta por múltiples procesos que colaboren entre sí, intercambiando información mediante ficheros o tubos, es un ejemplo de aplicación concurrente.	V
24.	Un navegador web con varios hilos (uno para refrescar la información mostrada, otro para obtenerla desde el servidor, otro para leer desde teclado/ratón,…) ejecutado en un ordenador con un procesador con un único núcleo, es un ejemplo de aplicación concurrente.	V
25.	La gestión de las comunicaciones resulta más natural y eficaz, pues permite simultanear el uso de la red con otras actividades, respecto a la programación secuencial.	V
26.	Aprovecha mejor los recursos (memoria, comunicación vía red, dispositivos de E/S, múltiples núcleos del procesador, etc.) y así las aplicaciones pueden ser más eficientes, respecto a la programación secuencial.	V

En una empresa se utiliza un baño mixto en el que pueden entrar tanto hombres como mujeres, pero con la condición de que simultáneamente sólo pueda haber personas de un único sexo. Además, el baño tiene una capacidad limitada de 3 personas.

```
monitor Baño {
    condition lleno, sexo_opuesto;
    int ocupantes=0, capacidad=3;
    boolean mujeres=FALSE;
    entry entra_adulto(boolean esMujer){
       if (ocupantes+1 > capacidad) lleno.wait();
       if (ocupantes>0 && mujeres!= esMujer){
          lleno.notify();
          sexo_opuesto.wait();
          sexo_opuesto.notify();
     }
    ocupantes++;
    mujeres = esMujer;
   }

   entry sale_adulto(){
         ocupantes--;
         if (ocupantes+1 == capacidad)
               lleno.notify();
         else  if (ocupantes==0)
                  sexo_opuesto.notify();
    }
}
```

Asumiendo que tanto hombres como mujeres invocan a los métodos del monitor según el protocolo *entra_adulto() … sale_adulto()*, y sabiendo que el monitor debe controlar que no se exceda de la capacidad del baño y que solamente puedan usarlo personas del mismo sexo al mismo tiempo:

27.	En el monitor se hace uso de la reactivación en cascada, es decir, lo que en Java se conoce como *notifyAll()* . *JUSTIFICACIÓN: Efectivamente, el sexo_opuesto.notify() que aparece en el método entra_adulto permite que un hilo pueda reactivar a otro, y el que ha sido reactivado pueda reactivar a otro más, etc., de forma que se produce una reactivación en cascada.*	V
28.	Con la variante de Lampson-Redell, pueden entrar adultos de distinto sexo en el baño. *JUSTIFICACIÓN: En este caso, como al notificar el hilo reactivado pasa a la entrada del monitor, compite con otros hilos. Podría darse la situación de que, si en el baño hay tres hombres y tenemos*	V

	a varias mujeres esperando en "sexo_opuesto", cuando el último hombre del baño salga, reactiva a una mujer y justo antes de que dicho hilo pueda continuar, entonces entra en el monitor un hombre. Ese hombre verá que ocupantes=0, por lo que entra y coloca mujeres=FALSE. Cuando le toca ahora a la mujer reactivada, ésta NO entra al monitor desde el inicio, sino desde donde ha sido reactivada, es decir, desde sexo_opuesto.wait(). Lo primero que hace es reactivar a otra mujer y, como hay un "if" en vez de un "while", no se comprueba la condición de si hay otros hilos en el baño de otro sexo, de modo que pasaría directamente al baño, y lo mismo ocurriría con las mujeres que se fueran reactivando al realizar sexo_opuesto.notify(). *Para que pudiera funcionar con esta variante es imprescindible que todas las condiciones estuvieran dentro de bucles while.*	
29.	Este monitor ofrece una solución correcta con la variante de Hoare. *JUSTIFICACIÓN: El monitor inicialmente parece que actúe de forma correcta, pues si por ejemplo llega primero un hombre, entrará en el baño y si llegan a continuación hombres o mujeres, los dos primeros hombres podrán entrar y las mujeres se suspenderán inicialmente en sexo_opuesto (si en el baño hay menos de 3 hombres), pero luego se suspenden en lleno (si ya hay tres hombres). Por su parte, los hombres se suspenden en lleno si ya hay tres hombres. Cuando el primer hombre salga del baño, notifica en lleno. Si quien se despierta es una mujer, entonces como no podría entrar (pues mujeres!=esMujer) entonces despierta al siguiente hilo de lleno y queda suspendida en sexo_opuesto. Si quien se despierta es un hombre, entraría en el baño.* *Hasta ahí, todo bien. El problema aparece cuando ya no quedan más hombres esperando y los que están en el baño van saliendo. El último de ellos despierta al hilo en sexo_opuesto. Imaginemos que aquí se han llegado a suspender varias mujeres (por ejemplo, cuatro mujeres). Al reactivar a una de ellas, lo primero que hace es sexo_opuesto.notify(), es decir, reactivar a otra mujer, y así sucesivamente. Recordemos que en Hoare, el hilo que notifica se va a una cola especial y tiene prioridad sobre los hilos que puedan llegar al monitor.* *Cuando ya están todas reactivadas, entrarían al baño sin problemas (pues la condición de la capacidad ya no se llega a comprobar). Y por tanto podríamos sobrepasar la capacidad del baño. Aunque nunca podríamos tener adultos del mismo sexo dentro del baño.*	F
30.	Con la variante de Hoare, se pueden usar indistintamente sentencias "if" o "while" en el método *entra_adulto*, obteniéndose el mismo resultado. *JUSTIFICACIÓN: En la variante de Hoare, en general, resulta indistinto utilizar if o while, pues cuando se notifica a un hilo quien continúa ejecutándose es precisamente el hilo recién reactivado, por lo que encontrará el monitor en el estado por el cual se le ha reactivado.*	V

En el análisis de la planificabilidad de un sistema de tiempo real crítico:

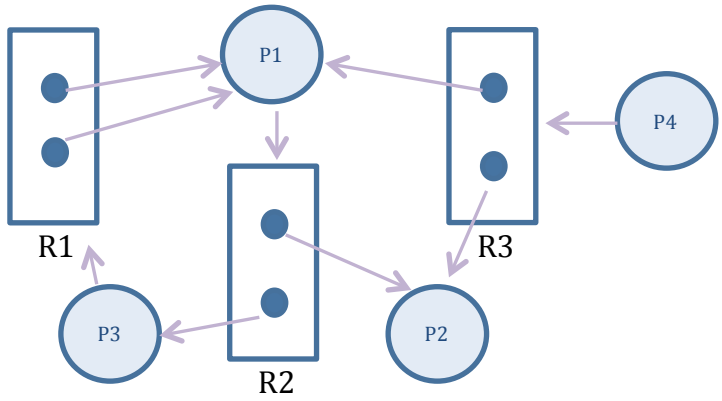
31.	Se calcula el tiempo de respuesta mediante una relación de recurrencia.	V
32.	Se calcula el tiempo de respuesta de cada tarea, según su intervalo de cómputo y las interferencias con tareas más prioritarias.	V
33.	Se asume un algoritmo de planificación basado en prioridades dinámicas no expulsivas.	F
34.	Deben cumplirse todos los plazos de respuesta de todas las tareas. Es decir, para cada una de ellas, la respuesta debe proporcionarse dentro de su plazo.	V
35.	Si se asume que las tareas no son independientes y se utiliza el protocolo de techo de prioridad inmediato, se debe incluir el factor de bloqueo en el análisis de planificabilidad.	V

Sobre la programación concurrente en Java...

36.	El método *yield*() de la clase Thread hace que un hilo pase del estado "ready-to-run" (preparado) al estado "blocked". *JUSTIFICACIÓN: Al realizar yield() se pasa de "Running" a "Ready-to-run"*	F
37.	Los objetos definidos en Java se mantienen en el 'heap' y son compartidos por los hilos que tengan una referencia a ellos.	V
38.	Para crear hilos en Java se debe utilizar el paquete *java.util.concurrent*. *JUSTIFICACIÓN: Ese paquete ofrece herramientas de alto nivel para la gestión de la concurrencia, pero para crear hilos basta con usar la clase Thread (o implementar la interfaz Runnable), ambas proporcionadas en la plataforma básica de Java.*	F

39.	El código a ejecutar por cada hilo debe estar contenido en su método *start*(), que puede reescribirse. *JUSTIFICACIÓN: El código a ejecutar debe estar contenido en el método run() de la clase Thread (que puede reescribirse), aunque para ejecutarlo se invoca al método start() de dicha clase.*	F
40.	Cada hilo Java tiene un nombre asociado, que puede modificarse con el método *setName*() y consultarse con el método *getName*().	V
41.	Si un hilo utiliza el método *wait*() de un objeto pasa del estado "running" (en ejecución) al estado "ready-to-run" (preparado). JUSTIFICACIÓN: Al realizar *wait()* se pasa de "Running" a "Waiting".	F

Dado el siguiente grafo de asignación de recursos…:



42.	Se puede afirmar que hay interbloqueo. *JUSTIFICACIÓN: Como hay al menos una secuencia segura, NO hay interbloqueo.*	F
43.	No existe ninguna secuencia segura. *JUSTIFICACIÓN: Un ejemplo de secuencia segura es: P2, P4, P1, P3.*	F
44.	Este tipo de grafos de asignación de recursos se utilizan en la estrategia de evitación de interbloqueos.	V
45.	Se trata de una asignación imposible, pues el proceso P1 tiene asignada más de una instancia del mismo recurso. *JUSTIFICACIÓN: Un proceso puede solicitar varias instancias de un mismo recurso.*	F

Se ha implementado la siguiente solución para una piscina:

```
public class PoolX  extends Pool0 {
      private boolean ok(int k, int i) {
           return (k<=log.maxKI()*i) &&  ((k+i)<=log.capacity());} // INVARIANT

       private int kids=0, inst=0, kidsWait=0;
       private void enterWait(int id)   {
        try {log.enterWait(id); wait();} catch (InterruptedException ex) {}}

       private void leaveWait(int id)   {
        try {log.leaveWait(id); wait();} catch (InterruptedException ex) {}}

       public synchronized long kidSwims(int id) {
              kidsWait++;
              while (!ok(kids+1, inst)) enterWait(id);
              kidsWait--;
              kids++;
              return log.swims(id);
       }
       public synchronized long kidRests(int id) {
              while (!ok(kids-1, inst)) leaveWait(id);
              kids--; notifyAll();
              return log.rests(id);
       }
       public synchronized long instructorSwims(int id) {
              while (!ok(kids, inst+1)) enterWait(id);
              inst++; notifyAll();
              return log.swims(id);
       }

       public synchronized long instructorRests(int id) {
              while (!ok(kids, inst-1)|| kidsWait>0) leaveWait(id);
              inst--; notifyAll();
              return log.rests(id);
       }
}
```

46.	El código propuesto permite resolver, al menos, todas las reglas asociadas a Pool2. *JUSTIFICACIÓN: En el invariante vemos que este código permite controlar que los niños no se queden solos y que no haya más niños por instructor de lo permitido.*	V
47.	El código anterior resulta adecuado para resolver las reglas asociadas a Pool3. *JUSTIFICACIÓN: En el invariante vemos que este código también permite controlar que no se supere la capacidad de la piscina.*	V
48.	El bucle *while* del método kidRests no es necesario, porque un niño nunca tiene que esperar para salir de la piscina. *JUSTIFICACIÓN: Los niños no tienen restricciones para abandonar la piscina. Los instructores son los que tienen las restricciones.*	V
49.	Se está dando prioridad a los instructores que esperan salir sobre los niños que quieren entrar. *JUSTIFICACIÓN: Los instructores que desean salir deben esperar tanto si no hay más instructores que se queden con los niños (por máximo niños por instructor), como si hay niños esperando a entrar (por capacidad de la piscina).*	F

Sobre la práctica 1 "Uso compartido de una piscina", cuyas reglas se muestran a continuación:

Tipo de piscina	*Reglas para N niños e I instructores*
Pool0	Baño libre (no hay reglas) *(free access)*
Pool1	Los niños no pueden nadar solos (debe haber algún instructor con ellos) *(kids cannot be alone)*
Pool2	Pueden nadar un máximo de N/I niños por instructor *(max kids/instructor)*
Pool3	No puede haber más de (N+I)/2 nadadores nadando simultáneamente (máximo aforo permitido) *(max capacity)*
Pool4	Si hay instructores esperando salir, no pueden entrar niños a nadar *(kids cannot enter if there are instructors waiting to exit)*

50.	En la piscina Pool1 no es necesario suspender a ningún hilo, al no existir sincronización condicional. *JUSTIFICACIÓN: Si un instructor desea salir, quedan niños en la piscina y no hay más instructores, entonces debemos suspenderlo.*	F
51.	Si escribimos *NotifyAll*() al final de todos los métodos de la piscina Pool1, en algunos casos se despertará a hilos que no era necesario despertar. *JUSTIFICACIÓN: Si un instructor desea salir pero no hay más instructores, lo habremos suspendido. Si un niño entra en la piscina y al final del método tenemos notifyAll(), entonces despertaremos al instructor de forma innecesaria, pues hasta que no llegue un nuevo instructor, no podrá abandonar la piscina (por lo que se despertará y se volverá a suspender a continuación).*	V
52.	La implementación de la piscina actúa como un monitor.	V
53.	No es necesario poner la etiqueta *synchronized* en los métodos de la clase Pool0 porque en esta piscina no hay reglas asociadas.	V

Sobre la práctica 2 "Los cinco filósofos comensales":

54.	La solución basada en la clase PhiloBothOrNone resuelve el problema de interbloqueos rompiendo la condición de no expropiación. *JUSTIFICACIÓN: Ninguna de las soluciones vistas en el laboratorio rompía esta condición.*	F
55.	En el programa de los filósofos, el tiempo de espera desde que un filósofo coge el primer tenedor hasta que coge el segundo no afecta a la probabilidad de interbloqueo. JUSTIFICACIÓN: Si no se ha aplicado ninguna solución al problema, se pueden producir interbloqueos. La probabilidad de que aparezcan es mayor conforme mayor sea el tiempo de espera entre que un filósofo coge su primer tenedor y su segundo tenedor, pues se aumenta la probabilidad de que otro filósofo coja el tenedor que el anterior necesitaba.	F
56.	Se ha comprobado que no existe ninguna solución totalmente correcta al problema de los cinco filósofos, pues siempre existirá algún caso, aunque sea poco probable, de interbloqueo. *JUSTIFICACIÓN: Las soluciones propuestas rompen alguna de las condiciones de Coffman y con ello se garantiza que nunca se podrán producir interbloqueos.*	F
57.	Una de las soluciones al problema del interbloqueo consiste en que los filósofos pares cogen primero el tenedor derecho y luego el izquierdo, mientras que los impares cogen primero el izquierdo y luego el derecho.	V

Dada la estrategia de coger los dos tenedores o ninguno, se proponen a continuación dos implementaciones distintas del método *takeLR*, en la clase *BothNoneTable*, que pretende coger los dos tenedores de forma atómica:

Opción 1

```
public synchronized void takeLR(int id){
    while (!log.rightFree(id)) { waiting();}
    while (!log.leftFree(id)) {log.wtakeLR(id);waiting();}
    log.takeLR(id);
}
```

Opción 2

```
public synchronized void takeLR(int id){
    while (!log.rightFree(id)) { log.wtakeL(id); waiting();}
    log.takeL(id);
    while (!log.leftFree(id)) {log.wtakeLR(id);waiting();}
    log.takeR(id);
}
```

58.	En la opción 2, mientras un filósofo espera que su tenedor izquierdo quede libre, otro filósofo le podría quitar el tenedor derecho.	V
59.	La opción 2 permite romper la condición de Coffman de retención y espera. *JUSTIFICACIÓN: La opción 2 no funciona de forma correcta, por lo que se podría producir retención y espera, pues el filósofo coge un tenedor y luego se pone a esperar.*	F
60.	En la opción 1, mientras un filósofo espera que su tenedor izquierdo quede libre, otro filósofo le podría quitar el tenedor derecho. *JUSTIFICACIÓN: Primero el filósofo espera al tenedor derecho, pero no lo coge. Luego espera al izquierdo. Cuando ya va a coger ambos tenedores (con log.takeLR), podría darse el caso de que le hubieran quitado el tenedor derecho.*	V