CDI Examen Teoría **Junio de 2018**

Apellidos: _____ **Nombre:** _____

D.N.I: _____ **Firma:** _____

Lo/as estudiantes **con** evaluación continua deben contestar solamente las primeras cuatro (4) preguntas (**Parte I**) y el tipo test (**Parte III**). Tiempo: 1 hora y media.

Lo/as estudiantes **sin** evaluación continua deben contestar todas las preguntas (Parte I+II+III). Tiempo: 3 horas y media.

Una vez empezado el examen, el/la estudiante acepta que cualquier uso de un dispositivo móvil sin previo aviso al profesor, resulta en un suspenso inmediato del examen con puntuación 0, informe a los órganos competentes de la Universidad de un posible intento de fraude, y posibilidad a un futuro examen en este curso solamente mediante orden escrita del órgano superior.

Parte I

Pregunta 1: [1 Punto] Durante el desarrollo del código para una aplicación distribuida que realiza procesamiento de imágenes (tal como hemos realizado algunas en las prácticas) intentas mandar un objeto tipo matriz por un `ObjectOutputStream` (usando un `socket` en el puerto 4445).
La parte de tu código responsable para el envio desde el cliente al servidor es:

```
...
Matrix matrix = new Matrix(4);
sd = new Socket(host.getHostName(),4445);
oos = new ObjectOutputStream(sd.getOutputStream());
oos.writeObject(matrix);
...
```

Cuando ejecutas tu programa (obviamente fue compilable) te encuentras con el siguiente error durante la ejecución:

```
java.io.NotSerializableException: Matrix
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
  at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
  at Client.run(client06.java:134)
```

Explica que es el problema y como debes actuar para resolverlo.

Pregunta 2: [1 Punto] ¿Cuál es el problema o cuáles son los problemas que se pueden presentar con el uso la siguiente clase en una aplicación concurrente?

```
class MyDouble {
  private double d;
  private ReentrantLock lock;
  MyDouble(double d) {
    this.d=d;
    this.lock=new ReentrantLock();
  }
  public void Div(Int I, Int J) {
    lock.lock();
    try {
      d=I.d/J.d;
      lock.unlock();
    }
    catch(Exception E) {
      d=0.0;
    }
  }
  public double Get() { return d; }
  public double Set(double d) {
    lock.lock();
    this.d=d;
    lock.unlock();
  }
}
```

Pregunta 3: [1 Punto] Enumera las condiciones necesarias para que se produzca un bloque entre procesos en un programa concurrente. ¿Por qué un/a diseñador/a de software concurrente debe conocer tales condiciones?

Pregunta 4: [1 Punto] Explica brevemente que es una condición de carrera. Añade un ejemplo.

Parte II

Pregunta 5: [1 Punto] En clase hemos visto un protocolo de entrada y salida a una sección crítica para dos procesos donde los procesos ejecutan simétrico (protocolo de Dekker).

1. Detalle el protocolo en pseudo código.
2. Comprueba la corrección de la exclusión mutua.
3. Razona sobre la política de justicia de este protocolo.

Pregunta 6: [1 Punto] Detalle tres implementaciones de clases diferentes como realizar (en Java) una clase `contador` concurrente (valores enteras de tipo `long`) donde varios hilos pueden incrementar el valor del contador concurrentemente. ¿Cuál, según los experimentos que hicimos en clase, usarías? y por qué?

Pregunta 7: [1 Punto] Durante la fase de depuración de un programa concurrente, de repente, después de haber introducido una salida al terminal más, es decir, una línea del estilo `System.out.println(...)`, el programa se bloquea. Si se borra de nuevo la línea con el `println`, el programa parece que funcione de nuevo correctamente. ¿Qué crees que problema todavía tiene el programa?

Pregunta 8: [1 Punto] Explica la semántica del modificador `volatile` de Java y su uso en programas concurrentes, entre otras, ¿qué tiene que ver con una relación *ha-pasado-antes*, es decir, qué garantías tiene el/a programador/a con el uso de `volatile` escribiendo y leyendo variables en su código? Añade ejemplos.

Pregunta 9: [1 Punto] Describe brevemente las clases `Executers`, `CountDownLatch`, y `ReentrantReadWriteLock` disponibles en los paquetes `java.util.conconcurrent` y `java.util.conconcurrent.lock`. Destaca en cada caso su semántica principal y su especial relevancia para su uso en programas concurrentes.

Parte III: tipo test (6 Puntos: marca todo lo correcto)

1. En java podemos decrementar una variable entero de tipo `long i` de muchas maneras diferentes, por ejemplo con las siguientes instrucciones: `--i;`, o `i=i-1;`, o también `i-=1`.
   (a) Ninguna operación con una variable tipo `long` es atómica.
   (b) No hay problemas, ya que todas las operaciones con variables enteras son atómicas, pero no es así con flotantes.
   (c) `--i` e `i--` son atómicos solamente si se declara `i` adicionalmente como `volatile`.
   (d) La atomicidad de la operación puede aparecer como tal durante largas fases de ejecución del programa hasta que en algún momento se manifeste como error, que puede ser difícil de detectar durante la depuración del programa.
2. El principio de la bandera es:
   (a) Un teorema que se puede comprobar, por ejemplo, mediante contradicción.
   (b) Sirve para comprobar si un protocolo con dos procesos garantiza la exclusión mutua.
   (c) Ayuda en distinguir entre espera activa y espera finita.
   (d) Es un algorithmo imprecindible para implementar protocolos de entrada y salida libres de bloqueos.
3. Una espera finita en un protocolo de entrada a una sección crítica tiene las siguientes ventajas:
   (a) Si para todos los procesos se puede garantizar una espera finita, entonces nunca se produce un bloqueo.
   (b) Una espera finita es fácil de implementar con los métodos `wait()` y `notify()` en java.
   (c) Si se implementa bien la espera finita entonces también está garantizada la exclusión mutua.
   (d) La espera finita es una propiedad deseable para un protocolo de entrada a una sección critica.
4. El protocolo de Dekker (quinto intento) de acceso a una sección crítica exhibe las siguientes propriedades:
   (a) Uno de los procesos participantes puede quedarse en inanición.
   (b) Uno de los procesos participantes tiene prioridad respecto al otro.
   (c) Ambos procesos se alternan necesariamente en el acceso al recurso si hay intentos de accesos simultáneos.
   (d) El proceso que cede el paso realiza una espera activa.
5. Los bloques sincronizados en java, es decir, `synchronized(obj) { ... }` tienen las siguientes características:
   (a) El codigo en el bloque solamente puede ejecutar un hilo, es decir, está garantizado la exclusión mutua.
   (b) Variables declaradas dentro de tal bloque son local al hilo.
   (c) Si varios hilos llegan al mismo bloque sincronizado, java no establece ningún orden de entrada en tal bloque.
   (d) Si se anida dos bloques con el mismo objeto y se ejecuta un `wait()`, el hilo libera los dos bloques.
6. La prevención para que no se produzca un bloqueo entre procesos en una aplicación concurrente:
   (a) Consiste en conseguir que no se cumpla ninguna de las 4 condiciones necesarias de un bloqueo.
   (b) Puede ser imposible en cierta aplicación.
   (c) Está implementado en los monitores de java ya por defecto, por lo menos si se usan solamente métodos sincronizados.
   (d) Es una tarea por estudiar (y conseguir, si procede), durante la fase de análisis y diseño.