CDI Examen Teoría **Mayo 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(s)] 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(s)] ¿Cuál es el problema que se presenta con el uso la siguiente clase en una aplicación concurrente?

```
class Int {
  private int i;
  Int(int i) { this.i=i; }
  synchronized void Add(Int I) { i+=I.i; }
  int Get() { return i; }
  int Set(int i) { this.i=i; }
}
```

Pregunta 3: [1 Punto(s)] Enumera las condiciones necesarias para que se produzca un bloque entre procesos en un programa concurrente.

Pregunta 4: [1 Punto(s)] Explica brevemente que es una condición de carrera.

Parte II

Pregunta 5: [1 Punto(s)] Razona críticamente sobre los tres métodos *prevenir*, *evitar* y *detectar/actuar* disponibles para gestionar posibles bloqueos entre procesos en una aplicación concurrente.

Pregunta 6: [1 Punto(s)] Detalle tres implementaciones de clases diferentes como realizar (en Java) un contador concurrente (valores enteras de 64 bit) donde varios hilos pueden incrementar el valor del contador concurrentemente.

Pregunta 7: [1 Punto(s)] Durante la fase de depuración de un programa concurrente, a la responsable de realizar las pruebas ocurre la idea de producir una versión del código introduciendo simples comandos de `sleep` de cierta duración justamente delante de todos los `wait` ya existentes en el programa. ¿Debería funcionar el programa lógicamente igual? En caso que sí, ¿qué tipo de fallos se pueden detectar, si el programa se comporta diferente? ¿Te ocurre algún inconveniente de tal prueba (a parte que las pruebas siempre consumen tiempo en la producción de una aplicación software)?

Pregunta 8: [1 Punto(s)] 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?

Pregunta 9: [1 Punto(s)] Describe brevemente cuatro (4) clases disponibles en los paquetes `java.util.conconcurrent` y `java.util.conconcurrent.lock`. Destaca en cada caso su semántica princial y su especial relevancia para su uso en programas concurrentes.

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

1. Un bloque sincronizado en java, es decir, `synchronized(obj) { ... }` tiene las siguientes características:
   (a) El codigo en el bloque solamente puede ejecutar un hilo, es decir, está garantizado la exclusión mutua.
   (b) Tales bloques no se pueden anidar.
   (c) Si varios hilos llegan al mismo bloque sincronizado, sus entradas en el bloque siguen un orden FIFO.
   (d) Si se anida dos bloques con el mismo objeto, el hilo se autobloquea.
2. Respecto a la espera finita en un protocolo de entrada se cumple:
   (a) Si para todos los procesos se puede garantizar una espera finita, entonces nunca se produce un bloqueo.
   (b) Una espera activa implica una espera finita.
   (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.
3. El protocolo asimétrico 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.
   (d) El proceso que cede el paso realiza una espera activa.
4. En java podemos incrementar una variable entero (long) `i` de muchas maneras diferentes, por ejemplo con las siguientes instrucciones: `++i;`, o `i=i+1;`, o también `i+=1`.
   (a) No hay problemas, ya que todas las operaciones con variables simples son atómicas, pero no es así con clases.
   (b) `++i` e `i++` son atómicos solamente si se declara `i` como `volatile`.
   (c) Ninguna operación con una variable tipo `long` es atómica.
   (d) La atomicidad de la operación está solamente garantizada cuando la variable `i` se manipula con hilos que se encuentran en el mismo procesador en un sistema distribuido.
5. El principio de la bandera es:
   (a) Un algoritmo más para implementar un protocolo de entrada y salida a una sección crítica.
   (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 criterio imprecindible para implementar algoritmos libres de bloqueos.
6. La prevención para que no se produzca un bloqueo entre procesos en una aplicación concurrente:
   (a) Siempre es posible.
   (b) Conlleva necesariamente el uso de FIFOs.
   (c) Está implementado en los monitores de java ya por defecto.
   (d) Es una tarea por estudiar (y conseguir, si procede), durante la fase de análisis y diseño.