CDI Examen Teoría y Práctica (Junio de 2020)

Versión C

Hay 3 versiones de examen, llamadas con los sufijos A, B, y C. Tienes que coger el examen según el ÚLTIMO **dígito de tu DNI: A: 1, 2, y 3; B: 4, 5 y 6; y finalmente C: 7, 8, 9 y 0**. Se te equivocas tendrás un no-presentado en esta evaluación.

Cada versión de examen contiene 3 apartados.

Los estudiantes en modo asistente deben trabajar apartado I (teórico, con preguntas tipo test respuesta única con penalización y preguntas de desarrollo corto) y apartado II (práctico, con preguntas tipo test de respuesta múltiple con penalización, preguntas de respuesta corta, y resolución/desarrollo de pequeños trozos de código). *Si ha superado por lo menos con un 4 el apartado I (teoría P1/P2) en mayo (es decir alcanzado por lo menos 8 de los 20 puntos), puede decidir no hacer apartado I, se usará la puntuación alcanzado en mayo. También puede decidir no hacer apartado II, se usará la puntuación (P3) alcanzado en mayo. P3 no tiene umbral mínimo, pero ojo, la nota ponderada final tiene que alcanzar por lo menos un 5.*

Los estudiantes en modo **no-asistente** y aquellos que no han superado en P4 (entregas a lo largo del curso) el mínimo de 4, deben trabajar **los tres apartados** de preguntas diversas.

La entrega del examen será a través de FaiTIC, subiendo vuestras respuestas, preferiblemente en un solo fichero, preferiblemente escrito por ordenador. Recomendamos usar un editor de texto (word, libreoffice, texto plano...) y contestar siempre anteponiendo la referencia a la pregunta. Para las respuestas tipo test basta comunicar el número de pregunta y la letra de la respuesta.

Es obligatorio estar conectado al campus remoto todo el tiempo.

Instrucciones detalladas para el examen:

(a) Accedéis a la **sala de exámenes 1** de la ESEI:
`https://campusremotouvigo.gal/access/public/meeting/106540688`

(b) Tenéis que identificaros con vuestro DNI y estar previamente registrados. Si no lo habéis hecho, registraos con anterioridad para evitar problemas de última hora. El sistema os busca por DNI en el conjunto de usuarios registrados en UVigo, por lo que ya completa vuestro nombre y apellidos.

(c) Entráis por el acceso de estudiantes con la clave **KuNh2xXV**

(d) Conectaos un rato antes: a las 15:50. Por favor, hay que estar también atento al correo por si os enviamos algunas instrucciones de última hora o enviadnos un correo (analia@uvigo.es, dnolivieri@gmail.com, formella@uvigo.es) si tenéis algún problema.

(e) Durante la realización del examen, deberéis tener en todo momento la webcam y el micrófono encendidos (por favor mantened vuestro entorno silencioso dentro de lo posible). Si no tenéis los medios técnicos para tal conexión, obviamente no los podéis usar, pero mirad el último punto.

(f) En el sistema hay dos tipos de usuarios: profesor y estudiante. Los profesores tenemos una pantalla en la cual os vemos a todos. El alumnado solamente ve al profesor de la sala.

(g) Al tener la webcam activada, os estaremos viendo todo el tiempo. Tenedlo en cuenta para elegir el sitio en el que estáis por la pérdida de privacidad que eso supone.

(h) En principio el sistema permite grabar lo que está ocurriendo en todo momento, pero eso es una funcionalidad que NO vamos a usar.

(i) Procurad que el sitio en el que estáis tenga buen acceso al internet. Si usáis Wifi compartida procurad también que no haya otros usuarios en la red haciendo un uso demasiado intensivo de la conexión.

(j) Si usáis portátil procurad que tenga batería llena y/o esté conectado a la red eléctrica.

(k) Si detectamos fraudes o tenemos sospechas en las entregas por haber copiado de otras personas, nos reservamos la posibilidad de ratificar los conocimientos mediante una entrevista oral a posteriori.

Apartado I (para todos, P1/P2)

A continuación se incluyen 14 preguntas (14 Puntos) tipo test. Cada pregunta tiene una sola respuesta correcta. La puntuación de las preguntas tipo test se obtendrá aplicando la fórmula:

$$(aciertos - 0{,}25 \cdot fallos)$$

Tipo test 1: Si se ha desarrollado una aplicación concurrente que funciona correctamente en un sistema actual y cambiamos el sistema por uno nuevo con otras características respecto al número de núcleos, tamaño de memoria caché, velocidad de reloj, y ancho de banda hacia la memoria entonces el nuevo sistema ejecutará la aplicación

(a) con una ganancia (*speedup*) que resulta más o menos igual que la suma de las diferentes mejoras empleadas.

(b) con un tiempo que puede ser incluso más lento dependiendo de los mecanismos de sincronización empleados en el programa concurrente a pesar de mejoras en las características.

(c) más rápido según el eslabón más grande de todas las mejoras empleadas.

Tipo test 2: En java se usan bastante los bloques `try-catch-finally`. Si se coloca dentro del bloque `try` un bloque sincronizado (es decir, con un `synchronized(obj) ...`), y se provoca una excepción (por ejemplo por división entre 0) entonces el monitor libera el cerrojo del objecto `obj`

(a) antes de entrar en el bloque `catch` correspondiente a la excepción.

(b) en cualquier momento una vez haber detectado la excepción.

(c) antes de entrar en el bloque `finally`.

Tipo test 3: El protocolo asimétrico de acceso a una sección crítica visto en clase exhibe las siguientes propriedades:

(a) Uno de los procesos participantes puede quedarse en inanición.

(b) Ambos procesos se alternan necesariamente en el acceso al recurso.

(c) El proceso que cede el paso tendrá una espera finita pero activa.

Tipo test 4: Para prevenir un bloqueo basta con garantizar que

(a) no se cumplan las cuatro condiciones necesarias.

(b) no se cumpla una de las cuatro condiciones necesarias.

(c) no se cumpla la condición más adecuada entra las cuatro condiciones suficientes.

Tipo test 5: ¿Cuándo vimos el mejor speed-up entre sistemas?

(a) Cuándo comparamos con el mismo número de hilos/procesos en uso.

(b) Cuándo medimos algoritmos que ocupaban casi solamente la CPU.

(c) Cuándo medimos algoritmos que mueven muchos datos por las características del problema.

Tipo test 6: ¿Cuál es lo correcto?

(a) Una espera finita implica una espera activa.

(b) Una espera activa implica una espera finita.

(c) Una espera activa no implica una espera finita.

Tipo test 7: El principio de la bandera es:

(a) Un algoritmo para implementar un protocolo de entrada y salida a una sección crítica que garantiza la exclusión mutua.

i Una herramienta para comprobar si un protocolo con dos procesos garantiza la exclusión mutua.

(b) Un criterio imprescindible para cumplir con la exclusión mutua entre dos procesos.

Tipo test 8: ¿Para los sistemas vistos en clase, por qué el sistema con procesador i9 es solamente más o menos el doble de rápido que el sistema con procesador i7 en el programa concurrente que implementa el algoritmo counting sort?

(a) Las velocidades de los relojes de ambos sistemas no permiten un speed-up más alto.

(b) La ley de Amdahl limita el speed-up más allá de los valores observados.

(c) Se observa con las mediciones donde está el cuello de botella durante la ejecución de este algoritmo.

Tipo test 9: La ley de Amdahl describe

(a) cual es el factor de acceleración alcanzable si se aumenta el número de procesos para un programa paralelo con carga de trabajo fijada.

(b) cual es el factor de acceleración alcanzable si se aumenta el trabajo por realizar para un programa paralelo con número de procesos fijado.

(c) cual es el factor de acceleración necesario para determinar si un problema es solucionable con un programa paralelo con cierto número de procesos participantes.

Tipo test 10: En un sistema donde medimos el tiempo de ejecución del programa de ordenación hemos observado la peor ganancia (*speedup*) entre ordenación estandar y ordenación por contado cuando el número de hilos empleados

(a) era uno, es decir, probamos los algoritmos paralelos con un solo proceso.

(b) era (más o menos) el número de CPUs disponibles en el sistema, es decir probamos en los sistemas justamente ocupados.

(c) era más o menos tan grande como el rango por ordenar, es decir, probamos los algoritmos en su escalabilidad.

Tipo test 11: En java se usan bastante los bloques `try-catch-finally`. ¿Si se coloca dentro del bloque `try` otro bloque `try-catch-finally` y se provocan dos excepciones, una en el bloque `try` anidado y otra en el bloque `catch` correspondiente a la primera excepción, cuando se ejecuta el bloque `finally` anidado?

(a) Al final cuando se ha tratado las dos excepciones en sus correspondientes bloques `catch`.

(b) Antes de la captura de la segunda excepción.

(c) No se ejecuta el `finally` anidado, ya que había otra excepción en la captura, solo se ejecuta el `finally` del bloque exterior.

Tipo test 12: Un bloque sincronizado en java, es decir, `synchronized(obj) { ... }` tiene la siguiente característica:

(a) Tales bloques sincronizados no se deben anidar ya que se ha obtenido el monitor previamente y por eso es ineficiente.

(b) Si varios hilos llegan al mismo bloque sincronizado, existe un orden predeterminado como entran en el bloque de código con exclusión mutua.

(c) Si se anidan dos bloques con el mismo objeto, y se ejecuta un `wait()` se liberan los dos bloques para que pueda entrar otro hilo.

Tipo test 13: ¿Por qué hemos visto en el sistema con el procesador Intel i9 (10 núcleos reales, 20 núcleos con hyperthreading) un factor de aceleración de 10 para la ordenación estandar, pero solamente un factor de 2 para la ordenación basada en contar (counting sort)?

(a) El algoritmo de ordenación estandar es mejor paralelizable que el algoritmo de counting sort.

(b) El algoritmo de counting sort es mejor paralelizable que el algoritmo estándar.

(c) El algoritmo de counting sort está limitado por el ancho de bando hacia la memoria desde la CPU.

Tipo test 14: Una condición de carrera (*race condition*) es

(a) una condición que se presenta cuando el orden temporal de dos o varios eventos no es correcto según requisitos de la aplicación.

(b) una condición que hay que cumplir entre procesos para garantizar la exclusión mutua de una sección crítica.

(c) una condición para llegar tan pronto como posible a un punto de sincronización con el fin de alcanzar un recurso.

Pregunta 1: [2 Punto(s)] La *región crítica* es un concepto abstracto útil para la programación concurrente. Este concepto está disponible en el lenguaje de programación Java. Detalla su síntaxis y uso. Razona críticamente sobre posibles ventajas y desventajas, tanto a nivel del propio concepto como herramienta de programación concurrente, como a nivel de implementación en el lenguaje Java. ¿Es fácil llevar el concepto a un entorno distribuido?

Pregunta 2: [2 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)? Están bien eligidos los lugares de insertar las demorras.

Pregunta 3: [2 Punto(s)] En clases hemos visto como ejemplo una implementación de una multiplicación de dos números con un programa concurrente. Empezamos con un código secuencial correcto. Con el simple uso de este código secuencial en los procesos participantes no logramos un programa concurrente correcto. ¡Explica cuales fueron las modificaciones necesarios para lograr finalmente un programa concurrente correcto y hasta cierta medida eficiente!

Apartado II (para todos, P3)

Problema 1 (Tema 2: Conceptos básicos)
Indique qué afirmaciones son verdaderas.

(a) Debido a que el orden de ejecución de los hilos depende de la JVM, un código multihilo en Java generalmente genera los mismos resultados independientemente del sistema.

(b) Cuando crea una clase de hilo extendiendo `Thread`, existe una gran flexibilidad para heredar otras clases.

(c) Cuando implementa la interfaz `Runnable`, debe implementar su método `run()`.

(d) Si su clase implementa la interfaz `Runnable`, entonces debe pasar su instancia al constructor de la clase `Thread`.

Problema 2 (Tema 3: Thread independencia)
Tu compañero de trabajo tiene que implementar un código utilizando varios hilos que ejecuta el `class Task`.
Sin embargo, antes de implementar todo lo que tiene que hacer `Task`, él quiere probar que todo funciona correctamente. En el método de `run()` de `Task`, él guarda el momento cuando comienza cada hilo en una variable `tiempoComienza` y imprime a la pantalla. Después de un `sleep()`, imprime este misma variable y el tiempo actual. A ver la salida mostrado abajo, está totalmente sorprendido.

```
Hilo 2 comienza:  Mon Jun 22 19:30:23 CET 2020
Hilo 3 comienza:  Mon Jun 22 19:30:25 CET 2020
Hilo 4 comienza:  Mon Jun 22 19:30:27 CET 2020
Hilo 5 comienza:  Mon Jun 22 19:30:29 CET 2020
Hilo 3  termina:  Mon Jun 22 19:30:29 CET 2020
Hilo 2  termina:  Mon Jun 22 19:30:29 CET 2020
Hilo 6 comienza:  Mon Jun 22 19:30:31 CET 2020
```

(a) Explique en palabras por qué está sorprendido.

(b) Proponer en palabras cómo arreglarías su código.

(c) Escriba el código necesario que podrá arreglarlo.

Problema 3 (Tema 4: Interrupts básica)
En el laboratorio 3.1(d), se le pide que use hilos para calcular PI, iterativamente. Sin embargo, también debe escribir un código en `Main` que interrumpirá el cálculo. (Para una pista, consulte la página 10 de las notas del tutorial de Lab 3, que proporciona un resumen de este código).

(a) Explique en palabras como enviar y detectar los interrupciones en este código.

(b) Escriba el código de la clase hilo Tarea que detectará la interrupción, de modo que todos los subprocesos finalmente se detengan.

Problema 4 (Tema 5: Locks)
Reemplace el siguiente clase `Contador` con una clase equivalente que use `ReentrantLock` en lugar de un bloque sincronizado:

```
1 public class Contador {
2   private static int counter; // initialized to 0 by default
3   public static synchronized int getID() {
4     int temp = counter + 1;
5     try {
6       Thread.sleep(1);
```

```
      }
      catch(InterruptedException ie) { }
      return counter = temp;
    }
  }
```

Problem 5 (Tema 5 y 6: wait/notify conditional)
En Practica 5, se le pidió que escribiese un código para el problema Enter/Wait, que actua como una sección crítica, (en un metodo compartido que se llama `Enterandwait()`) A continuación se proporciona un código para el método `run()` de un hilo que intenta ingresar a esta sección crítica si no ha estado dentro de la sección crítica justo antes. Explique si este código se compilaría y ejecutaría correctamente. Si no, indique qué está mal y proporcione el código que lo arregla.

```
public void run() {
  synchronized(miRef) {
    while(miRef.getNActual()>0) {
      while(miRef.getUltimo() == id) {
        try {
          System.out.println(this.id+" esperando...");
           wait();
        }
        catch (InterruptedException ex) {
          System.out.println(this.id+" interrumpido...");
        }
      }
      notifyAll();
      if(miRef.getNActual()>=1) {
        miRef.enterAndWait();
        miRef.setUltimo(this.id);
      }
    }
  }
}
```

Problem 6 (Tema 5 y 6: wait/notify conditional)
En el problema 5.2 de las tareas recomendada, se le pidió que escribiera el código para usar `wait/notfiy` para elegir el siguiente hilo que ejecutará `EnterAndWait()`. Modificamos esto para que solo se ejecuten hilos de colores (consulte la página 19, notas de laboratorio Lab06).

(a) Explique en palabras cómo se podría implementar esto llamando directamente al siguiente hilo (sin llamando a `notifyAll` a todos los hilos).

(b) Proporciona una implementación de código relevante, o segmentos de código, que mejor representa el método propuesto

Problema 7 (Tema 7: Thread-safe Buffer)
En el laboratorio 7, se le pidió que escribiera un código para una cola ”thread-safe”(sincronizado para hilos) que se puede utilizar con el patrón clásico de Productor/ Consumidor. A continuación se muestra una implementación de la clase utilizando Locks en uno de los métodos. Determine si este código es correcto. Si no, indique cual está mal y proporcione el código correcto.

```
class Buffer {
  int Capacidad = 10;
  List<Integer> lista = new LinkedList<>();
  final Lock lock = new ReentrantLock();
```

```
5     final Condition notFull = lock.newCondition();
6     final Condition notEmpty = lock.newCondition();
7     public int read() {
8       lock.lock();
9       try {
10        while(lista.size()!=0) {
11          try {
12            notEmpty.await();
13          }
14          catch (InterruptedException ex) {
15            Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE, null, ex);
16          }
17        }
18        notFull.signal();
19        return lista.remove(0);
20      }
21    }
22  }
```

Problema 8 (Tema 9: Java Concurrent Collections)
En el problema 2 de Lab08, se le pidió que diseñara un Productor/Consumidor con un Exchanger (ver apuntes de Lab08, página 19).

(a) Explique en palabras cómo se podrían implementar las funciones `put()` y `get()` con (java.util.concurrent.Exchanger)

(b) Proporcione una implementación del método `put()` utilizando un intercambiador para el problema del productor/consumidor.

Problem 9 (Tema 9: Código de Executors)
Dado este clase `CountingThreads`

```
1   public class CountingThreads  {
2     public static void main(String[] args)  {
3       Runnable r = new Runnable()  {
4         public void run() {
5           String name = Thread.currentThread().
6           getName();
7           int count = 0;
8           while (true)
9           System.out.println(name + ": " + count++);
10        }
11      };
12      Thread thdA = new Thread(r);
13      Thread thdB = new Thread(r);
14      thdA.start();
15      thdB.start();
16    }
17  }
```

(a) Explicar en palabras cómo funciona este código.

(b) Explicar en palabras cómo esto podría implementarse con los `Executors`

(c) Escriba una implementación en el código que mejor se adapte a su método propuesto.

Problem 10 (Tema 8: Executors)
Usando un *fixed thread pool*, dentro de un servicio `Executor`, queremos implementar un `Callable Task` que devuelve un `Future`. Para esto, responda a lo siguiente

(a) Explique en palabras cuál es la diferencia entre `Callable` y `Runnable` en un servicio `Executor`.

(b) Escriba una clase mínima, llamada `Task`, que use la interfaz `Callable` para devolver un valor entero y un variable `Future` que recoge este valor

(c) Escriba una clase mínima, llamada `Task`, que use la interfaz `Callable` para devolver un valor la suma de números dobles un variable `Future` que recoge este valor

Apartado III (solo no-asistentes)

Problema de prácticas
En Practica 10 y 11, se le pidió que implementara una arquitectura distribuida simple basada en el intercambio de mensajes. Esta consiste en un cliente/servidor que usa sockets y serialización para enviar objetos (mensajes y arrays) a traves de flujos (streams) entre diferentes clientes y un servidor central.
Como se explicó en el laboratorio, el servidor debe leer la imagen, asignar trabajo a los clientes que solicitan partes de la imagen, mantener una cola para aquellas partes de la imagen que se han procesado o necesitan procesarse, y finalmente volver a escribir la imagen total procesada. Para recibir mensajes en servidor, se puede implementar una clase `Server` Thread que solo maneje mensajes del cliente, mientras que otra clase `ServerData` Thread podría recibir partes de la matriz procesada de cada uno de los clientes.

(a) Mediante palabras, explique la arquitectura y cómo implementaría este problema en Java. Especifique algunos detalles esenciales de la lógica del programa, pero sin escribir todo el código.

(b) A continuación, responda las siguientes preguntas de implementación.

***a*)** Imagine que tiene una clase `MessageInfo` que se enviará entre cliente y servidor con los dos valores: `String code` y `int value`. Escriba la clase `MessageInfo` que se puede transferir utilizando *streams* entre el cliente y servidor.

***b*)** Escriba el segmento de código para la clase `Servidor` que espera un mensaje del cliente

***c*)** Escriba el segmento de código para la clase `ServerData` que espera las regiones de la matriz procesada de los clientes.

Preguntas de teoría

Pregunta 4: [2 Punto(s)] En clase hemos visto un protocolo de entrada y salida a una sección crítica para dos procesos donde los procesos ejecutan código diferente (protocolo asimétrico).

(a) Comprueba la corrección de la exclusión mutua.

(b) Razona sobre la política de justicia de este protocolo.

Pregunta 5: [2 Punto(s)] Explica que es una condición de carrera. Añade un ejemplo de tal condición en una aplicación y una solución para tu ejemplo.

Pregunta 6: [2 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 7: [2 Punto(s)] El manual de Java comenta que existe la posibilidad de que un hilo se despierta de un `wait` por razones desconocidas (*spurious wakeup*), es decir, el hilo termina el `wait` sin haber recibido un `notify`. ¿Qué solución propones para este posible problema que puede ocurrir en una máquina virtual de Java.