CDI Examen Teoría y Práctica (Junio de 2023)

El examen contiene **tres** apartados.
Las y los estudiantes en **modo asistente (evaluación continua)** deben trabajar el apartado I (tipo test, 10 puntos)) y el apartado II (preguntas de desarrollo, 10 puntos). **Tiempo 1.5 horas.**
Las y los estudiantes en **modo no-asistente** deben trabajar **los tres apartados**. **Tiempo 3.5 horas.**
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.

Apartado I (para todos, P1)

A continuación se incluyen 10 preguntas tipo test. Cada pregunta tiene una sola respuesta correcta (o más correcta), fallos no restan (10 Puntos).

Tipo test 1: En Java podemos incrementar una variable entera de tipo simple `int i` de muchas maneras diferentes, entre otras, con las siguientes instrucciones: `++i;`, o `i++;`, `i=i+1;`, o también `i+=1;`. Usando tales instrucciones en un entorno concurrente

(a) no provoca problemas, ya que todas los accesos a variables simples de 4 bytes son atómicas.

(b) solamente en caso de `++i` e `i++` son atómicos si se declara `i` adicionalmente como `volatile`.

(c) puede tener efectos no deseados ya que ninguna de tales operaciones con tal variable es atómica.

Tipo test 2: La ejecución de algoritmos concurrentes suele ser no-determinista dado que

(a) no se puede garantizar un orden temporal de los puntos de sincronización entre los diferentes procesos participantes.

(b) pueden ocurrir condiciones de carrera no deseadas a lo largo del tiempo de ejecución del programa en un sistema real.

(c) no se sabe de antemano en qué orden específico se van a ejecutar las instrucciones en el conjunto global de los procesos.

Tipo test 3: Las cuatro condiciones para que se produzca un bloqueo que hemos discutido en clase son

(a) condiciones necesarias.

(b) condiciones suficientes.

(c) condiciones necesarias y suficientes.

Tipo test 4: Cuando en un programa en Java se ejecuta el método `start()` de un hilo, el correspondiente `run()` empieza su ejecución de forma concurrente

(a) dependiendo si tal *start()* se encuentra dentro de un bucle, ya que en este caso se ejecuta primero todo el bucle dado que es la estrategia más eficiente.

(b) cuanto antes ya que la máquina virtual sabe que es un punto de comienzo de concurrencia y actua de tal manera ya por defecto.

(c) cuando toca según el planificador empleado en el sistema como combinación de actuaciones entre máquina virtual y sistema operativo.

Tipo test 5: El principio de la bandera es:

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

(b) Una herramienta para comprobar si un protocolo con dos procesos garantiza la exclusión mutua.

(c) Un criterio más para cumplir para garantizar la exclusión mutua entre dos procesos.

Tipo test 6: ¿Cuál crees es una mejor estrategia en el uso (lectura y escritura) de variables compartidas en un programa concurrente?

(a) Proteger las variables con cerrojos para cada acceso y evitar así posibles conflictos.

(b) Diseñar el programa de tal manera que un acceso concurrente en un mismo instante es improbable, así una protección de los datos con cerrojos no haría falta.

(c) Diseñar, si es posible, el programa de tal manera que un acceso concurrente en un mismo instante es solo de lectores o de un solo escritor, así una protección de los datos con cerrojos no haría falta.

Tipo test 7: La palabra reservada `synchronized` se usa para garantizar

(a) casos de uso de clases como objetos totalmente seguros para la concurrencia.

(b) transacciones atómicas con variables de sincronización.

(c) accesos con exclusión mutua a bloques de código o métodos de una clase.

Tipo test 8: 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 9: Una condición de carrera (*race condition*) es

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

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

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

Tipo test 10: ¿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.

Apartado II (para todos, P2) (10 puntos)

Este apartado tiene **4** (cuatro) preguntas para contestar.

Pregunta 1: [2.5 Punto(s)] Ha aparecido en el mercado un nuevo protocolo de entrada y salida para garantizar la exclusión mutua de una sección crítica involucrando dos procesos. Tienes acceso al código fuente y sabes que todas las instrucciones básicas se ejecutan de forma atómica en tu hardware. ¿Cuáles son las propiedades del protocolo que deberías analizar antes de comprarlo? ¿Entre ellas, cómo verificrías si el protocolo garantiza la exclusión mutua?

Pregunta 2: [2.5 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` con una duración aleatoria pequeña 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? ¿Se te ocurre algún inconveniente de tal prueba (a parte que las pruebas siempre consumen tiempo en la producción de una aplicación software)?

Pregunta 3: [2.5 Punto(s)] En clase hemos visto como ejemplo una implementación simple de una multiplicación de dos números con un programa concurrente basado en operaciones de incremento. 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 necesarias para lograr finalmente un programa concurrente correcto y hasta cierta medida eficiente!

Pregunta 4: [2.5 Punto(s)] Explica brevemente como se puede lograr en el sistema productor/consumidor que—después de cierto número de *servicios*—todos los productores y todos los consumidores terminaron sus tareas de forma adecuada (sin interrupciones) antes de terminar todo el programa.

Apartado III (solo no-asistentes) (20 puntos)

Este apartado tiene **8** (cuatro) preguntas para contestar.

Pregunta 5: [2.5 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 en Java. 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 6: [2.5 Punto(s)] Reemplaza la 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;
3    public static synchronized int getID(
4    ) {
5       int temp = counter + 1;
6       try {
7         Thread.sleep(1);
8       }
9       catch(InterruptedException ie) { }
10      return counter = temp;
11    }
12 }
```

Pregunta 7: [2.5 Punto(s)] En las prácticas, se ha pedido que se 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.

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

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

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

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

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

Pregunta 8: [2.5 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) Detalla en pseudo-código el protocolo.

(b) Comprueba la corrección de la exclusión mutua.

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

Pregunta 9: [2.5 Punto(s)] Razona críticamente sobre los tres métodos *prevenir*, *evitar* y *detectar/actuar* disponibles para gestionar posibles bloqueos entre procesos en una aplicación concurrente. Incluye una breve descripción de los mismos y un caso de uso.

Pregunta 10: [2.5 Punto(s)] ¿Para que sirve una `CyclicBarrier`? Explica su posible uso en programas concurrentes.

Pregunta 11: [2.5 Punto(s)] Explica que es una condición de carrera. Añade un ejemplo de aparición de tal condición en una aplicación concreta y una solución para tu ejemplo.

Pregunta 12: [2.5 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 programador o la programadora con el uso de `volatile` escribiendo y leyendo variables en su código?