Programación Concurrente y de Tiempo Real
Grado en Ingeniería Informática
Examen Final de Prácticas
Enero de 2015

Apellidos: Nombre:

Grupo (A ó B):

1. Notas

1. Escriba su nombre y apellidos con letra clara y legible en el espacio habilitado para ello.
2. Firme el documento en la esquina superior derecha de la primera hoja y escriba debajo su D.N.I.
3. Dispone de $2:30'$ horas para completar el ejercicio.
4. Puede utilizar el material bibliográfico (libros) y copia de API que estime convenientes. Se prohíbe el uso de apuntes, *pen-drives* y similares.
5. Entregue sus productos, utilizando la tarea de entrega disponible en el Campus Virtual, en un fichero (`.rar, .zip`) de nombre

   `Apellido1_Apellido_2_Nombre`

   que contendrá una subcarperta por cada enunciado del examen, la cual contendrá a su vez el conjunto de ficheros que den solución a ese enunciado en particular.

2. Criterios de Corrección

1. El examen práctico se calificará de cero a diez puntos y ponderará en la calificación final al 50 % bajos los supuestos recogidos en la ficha de la asignatura.
2. Las condiciones que una solución a un enunciado de examen práctico debe cumplir para considerarse correcta son:

a) Los ficheros subidos a través del Campus Virtual que conforman el examen práctico se ajustan al número, formato y nomenclatura de nombres explicitados por el profesor en el documento de examen.

b) El contenido de los ficheros es el especificado por el profesor en el documento de examen en orden a solucionar el enunciado en cuestión.

c) Los programas elaborados por el alumno, se pueden abrir y procesar con el compilador del lenguaje, y realizan un procesamiento técnicamente correcto, según el enunciado de que se trate.

d) Se entiende por un procesamiento técnicamente correcto a aquél código de programa que compila correctamente sin errores, cuya semántica dá soporte a la solución pedida, y que ha sido escrito de acuerdo a las convenciones de estilo y eficiencia habituales en programación

3. Enunciados

1. (Transformación Paralela de Matriz, 3.4 puntos) Dada una matriz cuadrada de datos númericos aleatorios $a_{i,j} \in [0, 1]$, se desea obtener una matriz transformada $b_{i,j}$ a partir de la siguiente ecuación:

$$b_{i,j} = 3 \cdot a_{i,j} + (a_{i,j-1} + a_{i,j+1} + a_{i-1,j} + a_{i+1,j})$$

Se pide desarrollar un programa que resuelva este problema de forma paralela especificaciones siguientes:

- La matriz tendrá una extensión de 1000×1000 definida mediante una constante de programa.
- El programa debe particionar el trabajo entre tantas tareas `Runnable` como núcleos haya disponibles de forma automática, en base a la ecuación de Subramanian y el coeficiente de bloqueo que estime para el problema. Puesto que el número de tareas será invariable, deberá procesarlas mediante un ejecutor adecuado a esa invariabilidad.
- Si considera que su solución requiere control exclusión mutua o sincronización, deberá proveerlas mediante objetos de clase `Semaphore`.
- El programa tendrá menú de usuario con dos opciones: la primera generará automáticamente la matriz de 1000×1000 elementos con números aleatorios. La segunda permitirá al usuario definir la dimensión de la matriz que desee, e introducir los datos iniciales por teclado, y además mostrará el resultado final en pantalla.
- Todo el código necesario residirá en el fichero `tParalelaMatriz.java`.

2. (Integración Definida Remota, 3.4 puntos) Se desea utilizar la técnica de integración numérica de Monte-Carlo para evaluar remotamente[1] la integral

$$\int_0^1 x^2 dx \tag{1}$$

[1]Recuerde y tenga en cuenta en su implementación que la arquitectura RMI realiza *multithreading* en el lado del servidor de forma automática y transparente para el programador.

Se pide escribir una arquitectura RMI distribuida capaz de hacerlo, de acuerdo a los requerimientos siguientes:

- Los ficheros `iRMC.java`, `sRMC.java` y `cRMC.java` contendrán la interfaz, servidor y cliente.
- El servidor y *dns* correrán en el puerto 2020.
- La interfaz definirá al menos un método donde se parametrice la precisión del cálculo remoto descrito en la ecuación (1) indicando el número de puntos que la técnica de Monte-Carlo emplea, y otro que permitirá hacer *reset* para iniciar un nuevo cálculo.

3. (Análisis de Rendimiento en Java, 3.2 puntos) Deseamos verificar de forma experimental la diferencia de rendimiento entre diferentes primitivas de sincronización en Java. Esta vez se van a comparar regiones críticas de Java implementadas con `synchronized`, cerrojos de clase `ReentrantLock` y objetos de clase `AtomicInteger`. La comparativa debe, utilizando un recurso compartido que modele a un contador, implantar acceso protegido al mismo mediante tareas que implementen la interfaz `Runnable`. Habrá 100 tareas definidas por implementación de la interfaz `Runnable`, que deberán ser procesadas mediante un ejecutor `FixedThreadPool`. El programa irá incrementando progresivamente (por ejemplo de 1000 en 1000) el número de accesos que cada tarea efectúa al recurso compartido concurrentemente con las demás, y ofrecerá una salida tabular similar a la siguiente, donde la primera columna recogerá el número de iteraciones por tarea, y las siguientes los tiempo totales con `synchronized`, `ReentranLock` y `AtomicInteger`.

Iter/Tarea	synchronized	ReentranLock	AtomicInteger
dato 11	dato 12	dato 13	dato 14
...	...	...	...
dato n1	dato n2	dato n3	dato n4