Programación Concurrente y de Tiempo Real
Grado en Ingeniería Informática
Examen Final de Prácticas
Febrero de 2014

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. Recuerde que la compilación con C++11 puede requerir el uso de `-std=c++0x` en lugar de `-std=c++11`.
6. Recuerde que los ordenadores están configurados para arrancar sin memoria de la sesión anterior. Por ello, recuerde que si cambia de sistema operativa es posible que pierda todo el trabajo hecho. Tengalo en cuenta. Recomendamos desarrollar todo el examen bajo sistema operativo Linux.
7. 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.

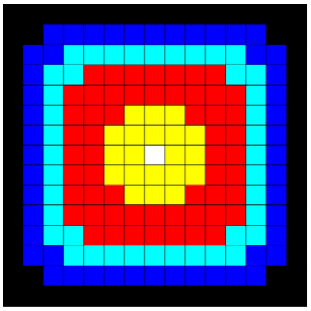
   b) El contenido de los ficheros es el especificado por el profesor en el documento de examen en orden a solucionar el enunciado en cuestión.

   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. (Ecuación de Calor Simple, 3.4 puntos) La ecuación de calor simple describe cómo cambia la temperatura a lo largo del tiempo, a partir de una distribución de calor inicial. Para un cuerpo plano y rectangular como el de la Figura



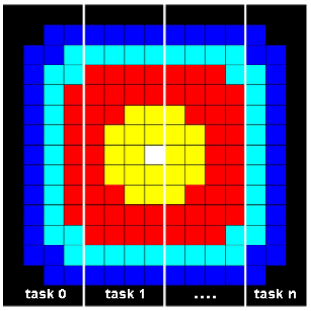
En ella, el núcleo central tiene alta temperatura y la frontera está fría. La temperatura entre los puntos del cuerpo va cambiando de acuerdo a la siguiente ecuación:

$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2U_{x,y}) + Cy * (U_{x,y+1} + U_{x,y-1} - 2U_{x,y})$$

done C_x y C_y son coeficientes que dependen de la naturaleza del cuerpo. Se pide desarrollar un programa que resuelva este problema de forma paralela (ver Figura inferior) de acuerdo a las especificaciones siguientes:

- Tendremos un rango de temperaturas entre -100 y 100 grados, y los coeficientes se leerán por teclado.

- El cuerpo tendrá una extensión de 1000×1000.
- Distribución de calor inicial: El límite externo del cuerpo estará a 100 grados y un núcleo central de 20×20 estará a 50 grados. El resto del cuerpo estará a 0 grados.
- El programa ofrecerá un menú de usuario con dos opciones. Una para procesar el cuerpo con la dimensión y la distribución inicial de calor ya indicadas. Otra, permitirá al usuario determinar la dimensión del cuerpo y la distribución de calor que crea oportuna, permitiendo mostrar el resultado final en pantalla.
- El programa debe particionar el trabajo entre tantas tareas `Runnable` como núcleos haya disponibles de forma automática. Puesto que el número de tareas será invariable, deberá procesarlas mediante un ejecutor adecuado a esa invariabilidad.
- Todos los cálculos se harán sobre un cuerpo (*array* de datos único). En consecuencia, deberá proveer exclusión mutua cuando haga falta. Para ello, utilice cerrojos de clase `ReentrantLock`.
- Todo el código necesario residirá en el fichero `eCalorParalela.java`.



2. (Guerra Galáctica, 3.4 puntos) El planeta *Helión III* orbita en el sistema estelar *Ophiuci* 36 perteneciente a la constelación de *Ophiucus*, y tiene la curiosa forma de un toroide reticulado de 200x100. En él habita una raza alienígena insectoide, que disemina sus 2500 mega-ciudades por el planeta, y que se prepara para invadir la Tierra. Esta civilización obtiene toda su energía de un miniagujero negro situado en una coordenada desconocida. Inteligencia Espacial Terrestre ha conocido los planes de invasión, lo cuál ha permitido a los *Space Rangers* lanzar un ataque preventivo coordinado contra el planeta. Inteligencia ha determinado que una destrucción de 2400 ciudades garantiza la seguridad, pero también se desconocen sus coordenadas respectivas. Por tanto, el ataque terrestre lanzará ojivas nucleares de clase 4 contra el planeta de forma aleatoria, y continuará mientras no reciba una petición de armisticio del enemigo (que este enviará cuando solo le queden cien ciudades) o cuando el planeta sea destruido por completo, si una de las ojivas impacta en el mini-agujero negro. Se pide escribir una arquitectura RMI distribuida que modele esta guerra galáctica, de acuerdo a los requerimientos siguientes:

- Ficheros `iHelion.java`, `sHelion.java` y `cHelion.java` contendrán la interfaz, servidor y cliente.

- El servidor correrá en el puerto 2020.
- El servidor irá mostrando en pantalla el resultado de los impactos que recibe el planeta, incluida la petición de armisticio o la destrucción total cuanto se produzcan.

3. (Análisis de Rendimiento, 3.2 puntos) Deseamos comprobar de manera empírica con el lenguaje C++ que los objetos de clase `atomic` son más eficaces en términos de rendimientos que los cerrojos `mutex` sobre un recurso compartido de tipo `int`. Para ello, debe desarrollar en `analisisRendimiento.ccp` el código necesario para ilustrar este comportamiento. El programa creará los hilos necesarios mediante funciones lambda, e imprimirá una tabla de tiempos con la estructura siguiente:

Total de Accesos al Recurso	Tiempo con Mutex	Tiempo con atomic
dato 11	dato 12	dato 13
...	...	...
dato 61	dato 62	dato 63

Para ello, es necesario saber medir tiempos de ejecución en C++. En el siguiente código, le proporcionamos un ejemplo con todo lo necesarios para ello:

```
#include <iostream>
#include <chrono>
#include <ctime>

long fibonacci(int n){
    if (n < 3) return(1);
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();
    std::cout << "f(42) = " << fibonacci(42) << '\n';
    end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "Calculo terminado a las: " << std::ctime(&end_time)
              << "Tiempo: " << elapsed_seconds.count() << "s\n";
}
```