Ejercicio 1

Cuestionario sobre *threads* y concurrencia

1. Explica qué ventajas y desventajas tiene el paradigma de programación paralela (*shared memory*) frente al paradigma de programación distribuida (*shared nothing*).

En una aplicación paralela, todas las tareas simultaneas pueden, en principio, acceder al mismo espacio de memoria, sin embargo, en el caso de una aplicación distribuida, las diversas tareas simultáneas no pueden acceder al mismo espacio de memoria. El principal motivo de esto es que algunas tareas se están ejecutando en computadores separados físicamente, y la transferencia de datos se realiza a través de la red o, duplicando datos.

En el caso de los sistemas de memoria compartida, compartir datos a través de varios subprocesos concurrentes de un solo ejecutable es rápido y mucho más rápido que usar la red como en programación distribuida. Además, tener un espacio de direcciones de memoria único puede hacer que escribir el código sea más simple. Pero una desventaja de compartir la memoria, es que hay que tener especial cuidado en el diseño del programa para evitar que varios subprocesos cambien variables sin que otros lo sepan.

Un sistema de memoria distribuida suele ser muy escalable. Otra ventaja que tienen, es que cada uno puede acceder a su propia memoria de forma aislada sin preocupación de que otro proceso este modificando datos. Las desventajas de este tipo de programación es que hay que implementar una estrategia para mover datos y ubicar datos. También hay que recalcar, que no todos los algoritmos se asignan fácilmente a este tipo de programación.

2. Describe tres tipos de aplicaciones reales de programación paralela y tres tipos de aplicaciones reales de programación distribuida.

Programación paralela:
- Aplicaciones meteorológicas con mapas de patrones climáticos, ya que los patrones climáticos y meteorológicos requieren de computación paralela.
- Entretenimiento, para cargas gráficas esadas
- Simulaciones virtuales

Programación distribuida:
- Redes de telecomunicación
- Navegadores web
- Control de procesos en tiempo real

3. Escribe una clase en Java que permita a dos *threads* hacer una búsqueda paralela de un entero dentro de una LinkedList. Concretamente, un *thread* debe comenzar a hacer la búsqueda por el principio de la LinkedList y el otro por el final. Al terminar, se debe informar cuál de los dos threads ha encontrado primero (si lo ha encontrado) el número. Hay que poner el tamaño de la LinkedList y el número a buscar como atributos de la clase.

Pasos para la realización del ejercicio:

Al ejecutar el programa hay que pasar por parámetro el número que se quiere buscar. Se ha creado una clase *Cerca*. Al crear esta clase, le pasamos el número que hay que buscar y la medida de la LikedList. En el constructor se crea la LinkedList de dicha medida y se guarda el número que hay que buscar. Esta clase, tiene dos métodos. Una que busca por el principio de la LinkedList y otra que empieza a buscar por el final. En el momento en el que se encuentra el número, el *thread* pinta por pantalla el descubrimiento.

La creación de los dos *threads* es simple. Se crean dos *threads* donde al implementar el método *run()*, se le indica a uno que realice la búsqueda por el principio de la clase *Cerca*, y al otro la búsqueda por el final.

4. Escribe una clase en Java que permita hacer una búsqueda paralela de un entero dentro de un array. Concretamente, la clase da este método:

public static int cercaParallela(int aBuscar, int[] Array, int NumThreads)

Este método crea tantos *threads* como especifica NumThreads, divide el array Array en tantas partes como NumThreads, y da (copia) a cada *thread* la parte del array que le corresponde para que haga una búsqueda secuencial del valor a buscar. Si un *thread* ha encontrado el valor, el método devolverá la casilla dentro del array inicial y mostrará por pantalla su número de *thread*. De lo contrario, el método devolverá -1. Hay que poner el tamaño del array, el número de *threads* y el número a buscar como atributos de la clase.

Para ejecutar este ejercicio hay que pasar como argumento el número que se quiere buscar y el número de *threads* que se desean. Se crea un array con una medida que se puede cambiar, y se llama al método que realiza la búsqueda paralela.
Primero de todo se divide el array según el numero de *therads* que hay. Una vez hecho esto, se crean los *threads* y estos empiezan. Cada uno calcula cuánto tiempo ha tardado en realizar su búsqueda.

5. Modifica el ejercicio anterior para que los *threads* accedan al array vía memoria compartida. Compara los tiempos de búsqueda con el ejercicio anterior y justifica las diferencias. Experimenta con diferentes tamaños del array y diferentes números de *threads*.

Observaciones:
Numero a encontrar = 999

Ejercicio 4

Medida = 1000 threads = 10 -> 22700ns
Medida = 100000 threads = 10 -> 35600
Medida = 10000000 threads = 10 -> 61200

Medida = 1000 threads = 100 -> 1900ns
Medida = 100000 threads = 100 -> 72500
Medida = 10000000 threads = 100 -> 69200

Medida = 1000 threads = 800 -> 100ns

Medida = 10000 threads = 5 -> 92800ns
Medida = 10000 threads = 8000 -> 300ns
Medida = 10000 threads = 9500 -> 100ns

Ejercicio 5

Medida = 1000 threads = 10 -> 7200ns
Medida = 100000 threads = 10 -> 111600ns
Medida = 10000000 threads = 10 -> 77500

Medida = 1000 threads = 100 -> 1500ns
Medida = 100000 threads = 100 -> 103000
Medida = 10000000 threads = 100 -> 24700

Medida = 1000 threads = 800 -> 200ns

Medida = 10000 threads = 5 -> 37600ns
Medida = 10000 threads = 8000 -> 100ns
Medida = 10000 threads = 9500 -> 400ns

6. Justifica qué diferencia hay entre usar el método *run* y el método *start* de la clase *Thread*.

La diferencia esta en cuando un método llama a *start()* un nuevo hilo se crea y luego ejecuta el método *run()*. Si se llama directamente a *run()*, no se crea ningún subproceso y el método *run()* se ejecutará como una llamada a un método normal.

En *multithreading*, una de las diferencias más importantes está en que el método *start()* y *run()* es que no se puede llamar más de una vez, mientras que el método *run()* si.

7. Escribe un programa multithread en Java que ordena un array de forma recursiva utilizando el método *merge sort.* El *thread* principal crea dos threads y cada uno va creando dos nuevos threads para ir ordenando el array.

8. Compara el tiempo de ejecución del ejercicio anterior con una ordenación secuencial. Justifica los resultados.

Medida array 1000
Tiempo ejecución secuencial: 4ms
Tiempo ejecución multithread: 342ms

El algoritmo secuencial es mucho más rápido debido a que el otro primero ha de ir creando todos los threads en cada nivel, para luego desde el ultimo nivel ir ordenando hasta llegar al primero. Sin embargo en el secuencial va realizando llamadas recursivas que realizan la ordenación.