Material complementario 07/04/2020

Arno Formella

Este documento recoge más o menos lo que hubiese contado en clases presenciales a lo largo de la exposición paulatina de las transparencias. En clase normalmente descubro diferentes partes del contenido de cada transparencia poco a poco, para no sobrecargar con información de golpe, y para mantener un hilo de pensamiento. En las transparencias distribuidas todo viene de golpe, por eso recomiendo trabajar con ellas lentamente y ver este material complementario a la par.

Estarán incluidas preguntas (a veces sin respuestas) para animar a la reflexión sobre el tema y a la búsqueda de información adicional. Admás proporciono más referencias a la red y la bibliografía.

Ojo, las páginas mencionadas aquí siempre se refieren a los tochos que publico para cada semana. Puede ser (y es casi seguro) que en el documento global (que desarrollo en paralelo) la enumeración de las página va a variar, ya que se añaden transparencias en otros lugares.

Recapitulamos:

- Hemos implementado un simple (y correcto) algoritmo de multiplicación, inicialmente pensado para el caso secuencial, de forma concurrente.
- Hemos notado que no funciona, ya que el producto esperado solamente vemos *a veces*.

P169

Si miramos la clase `Int` que hemos implementado, notamos el problema en el método `Add`, ya que no se ejecuta de forma átomica, sino se divide en diferentes subpasos cuales, cuando se intercalan de una o otra manera, producen resultados diferentes (parecido al simple ejemplo de `Inc` de antes).

¡Visualizate las posibles instrucciones que un procesador puede ejecutar para conseguir tal `Add` y las intercalaciones que producen problemas!

P170

La herramienta de Java para garantizar la ejecución de forma atómica de una secuencia de operaciones es el `synchronized`. Ya lo vistéis en acción en las prácticas.

Importante es el término *exclusión mutua*, es decir, que un hilo/proceso puede actuar sobre ciertos datos sin que otros interfieren.

Pero existen otros formas para implementar tal necesidad de atomicidad (cada una con sus ventajas y desventajas y campos de aplicación).

Empezamos con el `sincronized`.

P171

Como ya comentado antes, solamente el acceso a variables con como mucho 4 bytes (32 bits) es, según especificación, atómico, es decir, leer tal variable de la memoria y escribirla en ella.

Eso no es el caso de las variables de 64 bits. Ahí pueden, ya que los accesos son de 32 bits, generarse valores de variables que a priori *no existen* en el programa.

Fijaros si un hilo ejecuta `a=pi` (o `a=3.14159265358979323846` y otro hilo ejecuta `a=e` (o `a=2.7182818284590452354`). Entonces puede ocurrir que la variable `a`, una vez haberse intercalado mal las dos escrituras de 32 bits para formar el valor de 64 bits, contenga una *mezcla* de los dos valores (y quiero ver la persona que detecta tal fallo a simple vista ya que aparece en el fondo de los dígitos detrás de la coma!).

Un remedio es la palabra `volatile`, pero a parte de garantizar lecturas/escrituras de 64 bits atómicas, tiene dos efectos que influen en la eficiencia del programa que vemos más adelante.

P172+P173

Pues no hay que añadir mucho más.

Mira también el tutorial de oracle: `https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html`

Dado que los constructores no se pueden declarar como `sincronized` hay que vivir con el problemilla que comentan al final de tal página del tutorial como *Warning*.

P174

Pues la nueva clase con tal `synchronized Add`.

Para compilar el programa usamos la versión modificada `MultiM`, y simplemente usamos el mecanismo de enlaces simbólicos para conseguir la implementación con la nueva clase, es decir, en el terminal tecleamos:

```
ln -sf Int_int.java Int.java
javac MultiM.java
```

y ya está.

¡Eso debes hacer tu antes de seguir leyendo!

Ejecutamos por ejemplo (como antes): `java Multi 1000 1000 1`
y me sale como salida:

```
1000*1000 with 1 threads
starting worker... 0
exiting... 0
1000*1000=1000000 ??
exiting...
```

Hasta ahí bien. Haz más pruebas y verás...:

`java Multi 1000 1000 10`
y me sale como salida:

```
1000*1000 with 10 threads
starting worker... 0
starting worker... 8
starting worker... 5
starting worker... 9
starting worker... 3
starting worker... 1
exiting... 1
starting worker... 4
starting worker... 2
exiting... 2
starting worker... 6
exiting... 6
exiting... 4
exiting... 3
exiting... 5
exiting... 8
exiting... 9
exiting... 0
starting worker... 7
exiting... 7
1000*1000=1003000 ??
exiting...
```

Todo el gozo en un pozo. La salida no es la esperada. ¿hemos hecho algo mal? pues hemos añadido una palabra `synchronized` para garantizar la modificación atómica de las variables. ¿Está roto el ordenador? Jaja, va a ser que no!

¡Observa bien el bucle!

```
while(q.Get()>0) {
  r.Add(p);
  q.Add(minusOne);
}
```

Tenemos otra vez el mismo problema de las intercalaciones malignas. ¿Lo ves?

Ten en cuenta que varios hilos ejecutan el bucle, es decir, algunos cogen quizá el `q` con valor mayor que cero, y entran todos aquellos en el bucle. Entonces todos aumentan `r`, pero quizá han entrado demasiados hilos!

¿Que remedios hay?

- ¿Basta con declarar el método `Get` también como `synchronized`?
  ¡Pues no! ya que no cambia nada en la semántica.
- ¿Podemos embeber todo el bucle en un bloque sincronizado (por ejemplo bajo control de la variable compartida `r`)? Pues sí, pero ya no hay concurrencia ninguna, el primer hilo que consigue entrar en el bloque hará todo el trabajo, y los demás meramente verifican que todo está hecho.
- ¿Entonces?

¡Tenemos que cambiar el algoritmo!

P175

Realizamos el bucle de tal manera que un hilo solamente entra si hay *suficiente* trabajo por hacer, es decir, cambiamos la condición del búcle a:
`while(q.Get()>n) {`
es decir, solamente se sigue si hay trabajo para todos.

Claro, para tener a `n` en el ámbito del hilo, tenemos que construir el hilo con tal información.

P176

Luego puede ser que quede trabajo por hacer, ya que ninguno de los hilos trabajadores entra en el bucle si no pueden entrar todos (¡observa eso en tu código: pasa también `n` en el constructor!), a veces, queda el producto demasiado corto!

Eso resolvemos en el hilo principal, es decir, una vez terminados todos los hilos trabajadores, el hilo principal mira la variable `q`, si no es cero, significa que los hilos dejaban trabajo, y lo termina él en un bucle correspondiente.

En resumen, el fichero `MultiC.java` contiene todas las modificaciones para obtener el algoritmo concurrente de la multiplicación simple. Observa los siguientes detalles:

- Se ha aumentado la clase `Mul` (hilo trabajador) con el número de participantes `n`.
- Se ha modificado la condición del bucle.
- Se ha suprimido el mensaje final, ya que queremos medir tiempos de ejecución (y las salidas a la consola serializan y consumen tiempo).

- Se ha aumentado con un bucle para repetir la operación (para apaliar efectos de medición, igual como lo hemos hecho en el caso de la ordenación!).
- Se usa la clase `Timer` para realizar las mediciones.
- Se ha añadido un *check* automático del resultado (aprovechando que Java puede multiplicar :-), en vez de imprimir el resultado para el usuario.

Todo el código está de nuevo en el fichero `mul.tgz`.

Eso es todo, haz unos gráficos variando el trabajo y el número de hilos (tal como lo hemos hecho para la ordenación).

La semana que viene seguimos.