Material complementario 14/04/2020

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*.
- La semana pasada he adelantado a los interesados una solución a los dos problemas principales del intento.
- Aquí las primeras 8 transparencias son repetidas, pero no voy a repetir los comentarios, ya que están en el material complementario de la semana pasada. Sin embargo:
- Hemos superado el problema de la no atomicidad de la operación `Add` declarándola como `synchronized`. Con eso garantizamos que como mucho un hilo modifica a la vez una de nuestras variables que usan los hilos de forma compartida.
- Hemos superado el problema de la comprobación del bucle con un cambio en el algoritmo: solamente se entra en el bucle si hay trabajo para todos los participantes, posible trabajo no realizado se ejecuta al final en el hilo principal.

- Observa, aqui hablamos de soluciones correctas, no nos interesan, por el momento, cuestiones de eficiencia.

P177

Aquí está resumido lo que también menciono arriba.

Ahora vamos a ver como se llaman estos problemas en aplicaciones concurrentes a un nivel un poco más abstracto.

P178

Entonces tal linearizabilidad (o *linearizability*) es la propiedad que puede tener un sistema concurrente que podamos asumir que las operaciones sobre datos compartidos se ejecutan de forma instantanea o atómica.

Ten en cuenta que realmente, a bajo nivel de hardware, puede ser que no sea así, pero el modelo hacia capas más altas es, que podamos asumir que son atómicas, digamos, que esté garantizado desde las capas más abajo.

Ten en cuenta que en la realidad aunque una operación sea atómica necesitaría tiempo para su ejecución.

Ten en cuenta que si todos los participantes compiten para el acceso a tales operaciones atómicas, puede ser que se serializa todo el programa concurrente y no tenemos beneficio respecto a un programa secuencial (usé antes el ejemplo de embeber el bucle en un bloque sincronizado).

Ten en cuenta que en la mayoría de sistemas ciertas acciones ya son atómicas a cierto nivel (por ejemplo acceso a variables de 32 bit en Java), otras no (por ejemplo acceso a variables de 64 bit en Java), y se suele ofrecer técnicas (embebidos en el lenguaje o mediante liberías/paquetes o lo que sea) para conseguir exclusión mutua.

Veremos pronto más cosas en Java (y a nivel abstracto en general).

P179

Seguramente (o no) conocéis el concepto de linearizabilidad desde las bases de datos, por lo menos no os sorprende que las transacciones de todas las cuentas y tarjetas funcionan bastante bien en el mundo de los bancos.

Fijaros un momento en la tarea de implementar un sistema de ficheros (digamos una cosa como *Napster* en aquellos momentos históricos (`https://en.wikipedia.org/wiki/Napster`) donde la gente compartía, digamos, cierto material digital...).

Es para daros una idea: y seguro cada vez, os van a contratar empresas que quieren implementar/mantener/ampliar herramientas concurrentes.

P180

Pues la serializabilidad (*serializability*) va un poco más allá, no solamente queremos que las operaciones individuales funcionen con exclusión mutua (atomicidad), sino queremos que una secuencia de operaciones con la participación de varios procesos se ejecuten correctamente.

Un modelo por seguir es que pensemos que todas las operaciones individuales son linerarizables y cualquier intercalación posible (que reflejaría cada una un programa secuencial) nos llega a un funcionamiento correcto (como definido antes).

P181

Aquí solamente digo, que tales conceptos de linearizabilidad y serializabilidad no son necesarios para conseguir sistemas concurrentes que funcionen bien según los requisitos que tengamos.

Pero para estas clases, y usando Java como vehículo, nos limitamos a este modelo.

P182

Este caso (doy esta asignatura desde hace tiempo) se ha dado en directo en tal año, y fue un ejemplo interesante (y divertido) por seguir, y para ver que pasa si uno/a no se tome en serio las cuestiones de concurrencia en sistemas distribuidos.

Como dice la transparencia, la empresa ya no existe, pero la historia sigue existiendo...

P183

Aquí siempre digo (como profe chulito): *no ha hecho todo los esfuerzos*, ya que no fueron a clases de CDI (que hay en muchas universidades).

P184

Es básicamente lo que han implementado (y es parecido al bucle nuestro donde tenemos condición y decremento con `q`).

Puede ser que cada línea sea atómica, pero el conjunto (secuencia) no es serializable!

P185

Una transparencia de transición, usamos ahora algunas cosas de los paquetes de Java...

P186

Pues usamos tal paquete (ya lo conocéis desde las prácticas) y usamos 5 posibilidades para implementar nuestro entero para la multiplicación. En la transparencia falta todavía el `LongAdder`, pero lo añado, en el código ya está.

Un poco en adelante vemos lo mismo (lo siento, otra vez) con otras posibilidades que son los cerrojos y semáforos.

P187

Pues nuestra clase `Int` implementado con `Integer` y método sincronizado.

P188

Pues nuestra clase `Int` implementado con `AtomicInteger`. No hace falta `synchronized` ya lo hace bien la clase.

P189

Pues nuestra clase `Int` implementado con `LongAdder`. No hace falta `synchronized` ya lo hace bien la clase.

P190

Pues nuestra clase `Int` implementado con `LongAccumulator`. No hace falta `synchronized` ya lo hace bien la clase.

Como véis, todas las clases un poco diferente. No hay otra, leer el manual y apañarse.

P191

El codigo está subido otra vez en `mul.tgz`.

Esta gráfica muestra como se compartan las diferentes implementaciones en el sistema (que me he montado en casa para teletrabajar cómodamente...)

Os pido que hagáis tres cosas:

1. Conseguir la gráfica para vuestro sistema.
2. Intentar interpretar la gráfica (ya teniendo la mía incluso con una comparativa entre las dos).
3. Subir a FaiTIC un pantallazo (*screenshot*) donde se ve en el fondo la ventana con vuestra conexión a FaiTIC, y en primera fila vuestra gráfica que hais conseguido.

Aclaro: solamente el pantallazo, la interpretación es para vos, ya miramos la interpretación la semana que viene.

P192-195

Son algunas aclaraciones para el uso de `synchronized` que quizá no se pueden encontrar tan rápido en los manuales, y luego lo de siempre ***RTFM*** (`https://en.wikipedia.org/wiki/RTFM`).

Y claro, programar con los propios dedos!