Material complementario 21/04/2020

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:

- Estamos usando el ejemplo simple de un algoritmo de multiplicación basado en sumar, para trabajar con diferentes conceptos de concurrencia.
- Hemos producido un algoritmo correcto superando dos problemas que tenía la algoritmo ingenuo (y incorrecto).
- Hemos hecho mediciones, pero no ganamos nada en los sistemas reales (vea siguiente resumen de vuestras curvas).
- Hemos usado 5 clases diferentes para implementar el concepto de exclusión mutua para garantizar que la operación `Add` funcione sin que los hilos interfieran uno al otro.
- Hemos entendido (junto con las prácticas) el concepto de usar el `synchronized` en Java.

Las curvas que hais medido

En las curvas que se han entregado, se puede observar lo siguiente:

- Practicamente nunca se ha visto una mejora en el tiempo de cálculo usando más de un hilo (hay algunas excepciones donde 2 hilos ganaron a 1 hilo). ¿Cuál será la razón y que podemos hacer?
- En la mayoría de las mediciones ganan (respecto al tiempo de cálculo observado) las clases de nuestro `Int` implementadas con `LongAdder` y `LongAccumulator`, que es un poco lo prometen estas clases.
- En la mayoría de las mediciones pierden (respecto al tiempo de cálculo observado) las clases de nuestro `Int` implementadas con `synchronized`. ¿Parece que tal sincronización es bastante costoso?
- Las excepciones de tales observaciones arriba he visto solamente en los caso de haber usado una virtualización de un sistema operativo sobre otro (creo todos un linux sobre windows).

Unos comentarios en general, ya que visto que pasaba en ciertos gráficos enviados:

- Es conveniente usar escala logarítmia en el eje x, ya que nos interesa ver bien la evolución de las curvas tanto en caso de usar pocos hilos como en caso de usar muchos hilos.
- Se debe procurar que se vean todas las curvas en el gráfico, gnuplot normalmente lo hace de forma automática. Si se ha cambiado, por alguna razón, esta visualización se puede ejecutar `set autoscale xy` y se vuelve a la visualización por defecto.
- 5 de las 47 entregas (y 130 posibles) no han visualizado las cinco curvas superpuestas y generadas con propias mediciones en sus ordenadores... ¿qué problema había?

P199

A partir de ahora levantamos el punto de vista un poco a niveles de cierta abstracción (seguimos usando Java como herramienta para dar ejemplos).

Hemos observado (tanto con la multiplicación en teoría como en los ejercicios de prácticas) que los principales problemas aparecen cuando varios hilos/procesos quieren/deben manipular recursos compartidos.

P200

Para proteger nuestros datos de accesos concurrentes (y posiblemente incorrectos) de escritura, tenemos basicamente tres formas de actuar (dos ya hemos visto).

El concepto de la exclusión mutua garantiza que solamente un proceso tengo acceso para manipular el recurso (lo vimos en las implementaciones de la operación `Add` en las clases `Int`).

El concepto del comportamiento idéntico que si varios procesos acceden a un recuros todos actuan de forma idéntica, eso significa que da lo mismo quien *gana* (lo vimos con la variable `is_sorted` en la comprobación de la ordenación en nuestro algoritmo de ordenar).

El comcepto de transacciones donde se garantiza que una transacción que manipula el recurso va a ganar, las demás acciones/transacciones de otros procesos al mismo no tienen efecto: este concepto viene de las bases de datos. No entraremos en muchos detalles de este comcepto, lo menciona aquí para que sepáis de su existencia, será para clases de concurrencia avanzadas.

Los interesados pueden echar un vistazo a `https://en.wikipedia.org/wiki/Software_transactional_memory` donde hay más enlaces como este concepto ya está disponible en diferentes lenguajes de programación. Basicamente consiste que *mágicamente* el sistema garantiza que un conjunto de acciones (remirad el ejemplo de flexcoin de antes) se ejecuta de forma atómica. Existen CPUs que dan soporte al concepto a nivel hardware. El nuevo estándar de C++ incluye ciertos aspectos del concepto.

P201

Para implementar la exclusión mutua (aqui a nivel alto), necesitamos un protocolo acordado de actuación (todos conocemos tales protocolos, por ejemplo en los servicios de baño, con las puertas, los cerrojos y la indicación si está ocupado o libre).

Como se puede implementar un tipo de cerrojo a nivel bajo veremos un poco más adelante, ahora empezamos con el concepto a nivel alto.

P202

Imaginaos que varios procesos quieren entrar en una sección crítica, es decir, un trozo de código que queramos que se excute por un solo hilo.

(Sería los mismo: Imaginaos que varias personas quieren entrar en un sitio crítico, es decir, un lugar donde queramos que se encuentre solamente una persona).

Todos los participantes ejecutarían tal protocolos de entrada y salida.

Obviamente tendremos muchas posibilidades de implementar los protocolos de entrada y salida, todos con sus ventajas y desventajas.

P203

Aqui vemos la implementación de la operación `Add` en nuestra clase `Int` usando este concepto usando la clase `lock` (cerrojo) de Java.

Fíjate que delegamos el protocolo de entrada y salida a los métodos para tal fin de la clase. La sección crítico en este ejemplo es el aumento de la variable `i`.

Más en adelante hacemos de nuevo mediciones.

P204

Dado que tal uso de conceptos es muy común en la programación concurrente se han establecido varios formas de expresar ciertas situaciones comunes que hoy en día suelen estar presentes en casi todos los entornos de programación (y sistemas operativos) para facilitar en desarrollo de programas concurrentes.

Una vez más recomiendo leer y estudiar detenidamente los manuales de las clases (o lo que sea) que usáis para implementar programas concurrentes sea lo que sea el entorno concreto.

P205

El concepto de la *región crítica* ya está directamente disponible en Java. ¡Nota que eso no es el caso de C++!

Lo que está escrito aquí en pseudo-código está en Java con los bloques (o métodos) sincronizados.

El papel de la variable compartida es una referencia a un objeto que comparten los hilos.

P206

No hay mucho que añadir, es como usamos tal mecanismo en Java, compilando el programa el compilador inserta automáticamente los protocolos de entrada y salida para controlar el acceso con exclusión mutua a las secciones críticas.

Observa el detalle: si llamos desde un bloque sincronizado a otro tal bloque o función del *mismo* objeto el acceso está dado directamente (ya que el hilo ya posee el cerrojo).

P207

Este segundo concepto de regiones criticas condicionales no está disponible directamente en Java, sino hay que implementar lo con las funcionalidades de `wait` y `notify` como lo vimos en la práctica 5.

La idea es que un proceso solamente entra en una sección crítica si se presenta una cierta condición (por ejemplo, se permite a un productor que ponga un producto en una cinta si hay por lo menos un sitio libre en tal cinta).

Este concepto está disponible implicitamente en el paquete de concurrencia de Java (digamos a nivel de librería) con `ArrayBlockingQueue`.

P208

Fijaos que si varios hilos (personas) están esperando para entrar en una sección crítica (o un sitio crítico) y adicionalmente se tiene que comprobar una condición, hay que tomar decisiones cual de aquellos que esperan puede/debe entrar como siguiente.

P209

Es una una idea de implementación. Se usa dos colas.

Podéis mirar en la documentación de Java como se puede implementar tal concpeto de región crítica condicional en Java `https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/Condition.html` aprovechando de la interfaz `Condition`. Como véis en el código ahí se usa las dos colas (una de espera cuando se adquiere el cerrojo y otra cuando se espera la condición).

También recomiendas usar tal `ArrayBlockingQueue` directamente y no implementar el concepto de forma nueva.

P210

Obviamente hay que tener en cuenta cuestiones de eficiencia cuando se implementa y usa estos conceptos. Colas de espera frenan el paralelismo que se obtiene, cuando menos un procesos espera, más concurrencia se obtiene.

P211

El semáforo es el siguiente concepto abstracto para que la usamos en diseños de aplicaciones concurrentes.

Aquí ña idea es que varios (es decir, pueden ser más que uno) procesos entran al mismo tiempo en una sección crítica.

Uso normalmente una sala con cierto aforo como ejemplo. En tal caso se podría implementar un semáforo en la entrada de la sala. Mientras hay una silla libre dentro de la sala, el semaforo está en verde y una persona puede entrar. Si el aforo está completo, el semáforo se tiene que poner en rojo y los demás que quierán entrar tienen que esperar. Si uno sale de la sala, el semáforo se tiene que poner de nuevo en verde.

Ojo: ¡aquí en las transparencias se usa `wait` y `signal` que no es directamente lo mismo que el `wait` y `notify` que ya hais visto en prácticas! Las dos funcionalidades en las clases de Java se llaman `aquire` y `release`.

De hecho, en los diferentes entornos de programación hay muchas formas como llamar a las dos funcionalidades principales de los semáforos.

P212

El `init` define la capacidad del semáforo.

En Java, obviamiente se hace con el constructor.

Nota: si usamos como capacidad 1 tenemos un semáforo binario que al final sería lo mismo que un simple cerrojo.

P213

Llamaríamos al `wait` (o lo que sea lo correspondiente en el entorno de programación concreto) para pasar por el semáforo y entrar en la sección crítica.

P214

Llamaríamos al `signal` (o lo que sea lo correspondiente en el entorno de programación concreto) para salir del la sección crítica y dejar pasar por el semáforo otros que posiblemente puedan estar esperando.

P215

Aquí está en pseudo-código como se puede usar un semáforo binario para acceder a una sección crítica, es decir, implementamos el concepto de la región crítica con el concepto del semáforo.

P216

Aquí está en código de JAvacomo se puede usar un semáforo binario, es decir, construimos con 1, para implementar la operación `Add` en nuestra clase `Int` para garantizar exclusión mutua.

P217

Si en un caso concreto tenemos solamente cerrojos simples (o semáforos binarios) disponibles, podemos implementar un semáforo general usando dos semáfors binarios y un contador.

Vamos a implementar tal semáforo para que veáis como se puede implementar conceptos cada vez más altos con lo que tenemos a nivel más bajo.

P218

Si cogemos de nuevo el ejemplo de la sala de arriba: el primer cerrojo/semáforo proteja la puerta, el segundo cerrojo/semáforo proteja el contador.

El protocolo de entrada aquí hace los pasos que ya vimos antes: se espera hasta la puerta esté abierta, se entra (observa que nadie más puede entrar, ya que la puerta de cierre, es *binario*), se accedo al contador con exclusión mutua, se decrementa, ya que hay una persona más en la sala, significa un sitio menos libre, si queda un sitio más libre, se abre la puerta, si liberar el contador.

P219

El protocolo de salida aquí hace los pasos que ya vimos antes: se accede al contador con exclusión mutua, se incrementa, ya que hay una persona que saldrá de la sala y dejará un sitio libre, se abre la puerta, si fue el primero que sale de la sala completamente llena (observa que el último que entró no abrió la puerta al entrar otra vez!).

P220

La implementación del pseudo-código hecha en Java, observa que usamos el cerrojo `ReentrantLock` como semáforo binario.

Con este semáforo podemos implementar otra vez nuestra clase `Int`.

Con estás tres nuevas formas, es decir, con cerrojos, con semáforos, y con nuestra propia clase de semáforo puedes reimplementar la clase `Int` de nuevo y el programa de multiplicación debe funcionar igual.

Puedes generar un gráfico ahora con las ocho curvas superpuestas.

Esta vez os añado solamente la gráfica de mi sistema, las tres clases deberíais hacer vosotros (bueno dos ya están en las transparencias).

P221

Una crítica al usos de las estructuras de datos concurrentes que estamos usando, como véis en el código, hay que tener mucha disciplina para poner los métodos en su sitio correcto para que todo funcione bien.

P222

Como hemos visto (y tal está en Java), las regiones críticas tienen salida solamente al fin del bloque (en Java la llave {), el código necesario inserta el compilador, por eso no podemos emular tal concepto de región crítica con un semáforo.

P223

La gráfica ahora con las 8 implementaciones. Interpreta lo que ves.