Material complementario 28/04/2020

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:

- La semana pasada terminamos con ver como se puede implementar un semáforo general con dos semáforos binarios (o cerrojos).
- Hemos usado está construcción propia para implementar otra vez la clase `Int`.
- Hemos visto (está aquí repetido en la primera transparencia) que nuestro algoritmo de multiplicación funciona, pero que esta solución (curva negra) es la peor respecto a tiempo de cálculo que hemos observado.

P224

Hasta ahora hemos visto que nuestro algoritmo simple de multiplicación (recuerda: es un ejemplo sencillo para visualizar ciertos aspectos de concurrencia...) es correcto y se puede implementar la exclusión mutua con diferentes herramientas.

Pero también hemos observado que el uso de varios hilos no aporta nada respecto a eficiencia.

Dado que todos los hilos compiten para el acceso a las variables compartidas, un hilo al final incluso es más rápido que todos juntos.

Eso podemos remediar en dividir el trabajo por hacer (incrementar `r` y decrementar `q`) de tal manera a los diferentes hilos que cada uno puede trabajar independiente de los demás.

P225

Está cambiado la clase del trabajador para que use variables locales tanto para `p` como para `q` (siendo simples `int`).

Inicializamos especialmente `q` solamente con la parte que tiene que *trabajar* este hilo (por eso dividimos por el número de hilos).

P226

Luego reescribimos el bucle de cálculo que tal manera que trabaje con las variables locales y añadimos la variable `local_r` que nos accumule el producto parcial de este hilo.

Observa bien: ahora el bucle no accede a ninguna variable compartida con otros hilos.

Una vez hecho el trabajo, el hilo accumula su suma parcial al resultado global `r`.

Al final solamente tenemos un acceso con exclusión mutua en el código del trabajador.

P227

También tenemos que modificar ligeramente el hilo principal: dado que en los hilos dividimos por n para calcular el trabajo por hilo, puede ser que alguna parte no se hace (si q no es divisible entre n).

Por eso aprovechamos que el hilo principal haga esta parte (por eso el módulo con n). Este hilo también lo hace con sus variables locales, y lo accumula después de la sincronización (también se podría hacer antes, no importa) con exclusión mutua en la variable del resultado.

P228

La gráfica ya es otra: observa que estamos 1000 veces más rápidos.

He aumentado el valor de q en la prueba por un factor 500, y un hilo es el doble de rápido (más o menos) que un hilo antes.

Si hay muchos hilos, parece que se sigue perdiendo...

P229

Esta gráfica visualiza el comienzo de la curva (transparencia anterior), es decir, para pocos hilos.

Se ve que desde 1 hasta más o menos 8 (el número de núcleos que tengo) se está ganando velocidad. Parece que los 6-8 CPUs ganan un factor 5 más o menos comparado con un solo hilo.

También vemos que en esta situación de pocos hilos no importa cuales de las implementaciones para la clase `Int` usamos.

Resumamos:

- Se debe usar un algoritmo concurrente correcto.
- Se debe usar exclusión mutua correctamente donde es necesaria.
- Se debe reducir el acceso a variables compartidas para obtener eficiencia.
- Se debe usar las herramientas de las librerías (parece que trabajo propio que mejore no es sencillo :-)
- Se debe dividir el trabajo entre las partes de tal manera que se puede actuar individualmente para conseguir buena eficiencia.

Parecen conclusiones bastante obvias, pero hay que diseñar las aplicaciones concurrentes para que lo hagan según los requisitos. La multiplicación presentado obviamente no es un caso real (mejor multiplicar directamente en vez de incrementar), pero visualiza bastante bien estos diferentes aspectos.

P230

Hablamos algo sobre *monitores* que es el concepto abstracto que se usa (entre otras en Java) para gestionar el uso de objetos de forma compartida entre procesos.

La idea principal es encapsular el control para evitar (dentro de lo posible) un uso incorrecto.

P231

Se comentan unos detalles de los monitores para visualizar su función.

En Java tales conceptos están embebidos explicitamente en el lenguaje, por ejemplo, con la palabra reservada `synchonized`, tal como hemos visto en diferentes prácticas.

P232

Con tales liberaciones temporales de la exclusión mutua se puede pasar el acceso con exclusión mutua entre hilos.

Con la llamada a `wait` se permite que el hilo actual dueño del cerrojo, lo libera para que otro (según política del planificador del monitor) pueda entrar.

Con la llamada a `notify` se despierta a otro hilo (con `notifyAll` en Java a todos los demás) para que intente adquerir el cerrojo de nuevo (es decir, despierta de su `wait`).

Observa los siguiente detalles importantes:

- Un hilos solo puede regresar del `wait` si hay otro hilo que lo notifique. Eso es importante ya que complica la programación concurrente si queremos que el programa funcione también con un solo hilo (recuerda que tanto la ordenación como la multiplicación funcionaban correctamente con un solo hilo).
- Si hay varios hilos concurrentemente en su `wait`, depende de la política del monitor quien será el siguiente que actue como notificado (recuerda que en Java puede ser cualquier hilo).
- Hay que tener cuidado con la notificación ya que tiene que ser después (en el sentido causal) de que algún hilo esté en su `wait`, ya que si no, la notificación se pierde.
- Para que se pueda garantizar de se notifique en Java un hilo en concreto, este hilo debe ser el único que esté en su `wait` sobre tal objeto.
- Nota que el hilo adquire el cerrojo de nuevo, cuando el hilo que notifica libera el cerrojo, o bien llamando de nuevo a un `wait` o saliendo del bloque sincronizado.

P233

Existen alternativas para el concepto de monitores basado en cerrojos, de los cuales hablamos brevemente en la útlima clase.

Como hemos visto en el ejemplo de la multiplicación, el uso de `synchronized` es costoso. El `AtomicInteger` (y otros) tenían un mejor rendimiento, ya que están basados en este concepto de realizar concurrencia sin cerrojos.

Hay una empresa (spin-off de la Universidad de Chalmes) que ofrece productos para concurrencia de este estilo (los que quieren echar un vistazo: `http://www.non-blocking.com/`). Seguramente hay más, y habrá más soluciones en el futuro.

P234

Se enumera algunas de las posibles desventajas de cerrojos tal como están implementados en Java (con el mecanismo de `synchronized` y los monitores).

Observando y estudiando las posibilidades que ofrece el paquete de `java.util.concurrent` se ve que se ha aumentado las posibilidades de Java con el uso de las clases y estructuras de datos del paquete. Recuerda que tal paquete fue introducido en Java 1.5, las versiones anteriores solamente tuvieron el monitor.

Se menciona aqui especialmente en problema de las condiciones de carrera.

Una condición de carrera se presenta cuando tenemos que garantizar una secuencia específica de eventos (realizados por varios hilos diferentes) y no se cumple por alguna razón en la aplicación implementada.

Nota: la propia secuencia de eventos está dado por los requisitos de la aplicación (por ejemplo primero se abre un fichero, luego se escribe en él), su cumplimento durante la ejecución de forma concurrente

tiene que garantizar la implementación (hay que garantizar que el fichero esté abierto para tal hilo que quiere escribir).

Es bastante difícil comprobar que un programa no tenga condiciones de carrera (y es un típico fallo en sistemas operativos y sistemas de seguridad que se remienda muy a menudo con actualizaciones).

Un ejemplo de estos días: `https://www.zdnet.com/article/symlink-race-bugs-discovered-in-28-antivirus-products/` comenta sobre la existencia de tales problemas en herramientas antivirus :-)

P235

La máquina de estados de los hilos en Java.

Hay que tener este dibujo en mente cuando se implementan aplicaciones concurrentes en Java.

Nota adicional: como mencionados en prácticas existe el spurious wakeup (es una concesión a la realización de máquinas virtuales de Java y los sistemas operativos), que *no* está mencionado en el dibujo.

Para garantizar que un hilo solamente actua cuando fue notificado se tiene que tratar/gestionar tal información en la propia capa de la aplicación, por ejemplo, con variables de estados.

P236

Ahora salimos de la capa alta con estructuras de datos, conceptos, y lenguajes de programación y miramos como podemos implementar exclusión mutua con instrucciones básicas de la CPU.

Obviamente eso no hace falta mucho en la realidad de un programador o de una diseñadora de software, pero es instructivo ver como se hace y por qué funciona.

Veremos solamente unos ejemplos para dos procesos (ya nos llega :-)...

P237

Es un protocolo asimétrico que resuelve el problema.

Está conocido también como el protocolo entre vecinos para que un perro y una gata (que no sean buenos amigos) puedan acceder al patio común.

Podeis imaginar que las variables `v0` y `v1` son banderas que se ponen en las ventanas (`store` de un booleano) y se puede observar desde lejos (`load` de un booleano).

Recomiendo jugar paso por paso este protocolo (mejor con algo en las manos o en la mesa para ver bien el estado de las variables).

P238

Toca comprobar formalmente que el protocolo de antes garantiza la exclusión mutua.

Por eso usamos este teorema (es un teorema igual como, digamos, el teorema de Pitágoras), antes de usar lo, hay que comprobar el teorema.

P239

La comprobación funciona mediante contradicción.

Primero, observa que el teorema es simétrico respecto a los dos procesos, por eso, podemos asumir que cualquier de los dos es el último en mirar. Y P0 no ha visto la bandera!

Segundo, tendrá levantado su bandera.

Tercero, podemos asumir que no ha visto la bandera, ya que en caso contrario ya hemos terminado.

Cuarto, pues ya que los dos levantan, en algún momento también P1 lo tiene que hacer.

Quinto, P1 mirará después,

Sexto, entonces P0 no era el último en mirar, que es lo que asumimos al principio.

Nos sirvirá este teorema para comprobar ciertas propiedades de los protocolos.

P240

Son bastante intuitivos estás propiedades que debe cumplir un protocolo de entrada y salida a una sección crítica.

A veces, uno se olvida del segundo, y a veces esto provoca que una aplicación se queda bloqueada.

Es un hecho bastante obivio: si se puede garantizar que nunca ocurre una espera infinita, o en otra palabras, todas las esperas son finitas, entonces nunca hay bloqueo.

Tampoco hay que obviar el último: la sección crítica a su vez siendo código secuencial, no debe contener bucles infinitos.

P241

Si nos paramos un momento en el protocolo de entrada donde varios procesos compiten por el siguiente acceso, podemos analizar sus propiedades.

Se habla de justicia, pero solamente para tener un nombre para tal propiedad, no hay que confundirlo con la justicia entre humanos. Son meramente los requisitos que queramos que cumpla el protocolo.

Por ejemplo, el protocolo asimétrico que ya vimos antes tiene la la propiedad de justicia que uno de los dos participantes siempre cede en caso de intento de acceso simultáneo. ¡Es un buen momento de revisar el protocolo en este momento!

También vemos que si el proceso que no ceda se sincroniza, digamos, maléficamente, con el otro, puede conseguir que el otro nunca entra en la sección crítica. ¡Verifícalo en el protocolo!

La tolerancia a fallos menciona aquí porque puede ser importante en ciertas aplicaciones, el protocolo que vimos no tiene ninguna tolerancia a fallos, si un proceso no termina su sección crítica el otro nunca entrará. Claro inicialmente es el objetivo, pero existen situaciones donde hay que ser más, digamos, *flexible*. No entramos más en este asunto.

P242

Son las preguntas que uno se debe hacer para cualquier protocolo de entrada y salida para conseguir exclusión mutua.

Intenta de nuevo contestarlas respecto al protocolo asimétrico.

P243

En general los protocolos con solamente `load` y `store` (él que vimos y él que veremos a continuación) son bastante complejos, sobre todo cuando se aumenta el número de procesos participantes.

Por eso, ya temprano en el desarrollo de micro-procesadores con su conexión a la memoria, se ha inventado instrucciones hardware que facilitan la implementación de tales protocolos para conseguir exclusión mutua en presencia de más actores.

Las instrucciones atómicas a nivel hardware necesitan memoria común entre los participantes para que funcionen (y es una de las razones porque siguen siendo limitados en su eficiencia dado que no se pueden aprovechar de los cachés individuales de los sistemas).

El conocimiento de las posibilidades con solamente `load` y `store` también sirve para implementar los protocolos en sistemas distribuidos donde en vez de memoria común se tiene el envío de mensajes (`load` corresponde a `read` y `store` corresponde a `write`).

P244

Veremos el desarrollo de otro protocolo en cinco pasos para finalmente obtener un protocolo simétrico sin diferencia en justicia para los dos participantes (recuerda que el protocolo asimétrico tenía una justicia con prioridad para un lado).

Este primer intento asume que al principio que la variable `v` es o bien igual a `P0` o bien a `P1`.

Invierta un momento de tiempo para entender el protocolo simple.

Seguramente te recuerda a un simple ping-pong.

P245

Por el control mediante la variable `v` que especifica claramente quien puede entrar y quien no, y la condición que se cambia `v` solamente fuera de la sección crítica, tenemos la exclusión mutua garantizado.

Como bien se observa, los procesos solamente pueden entrar en su sección crítica de forma alterna, es decir, como en el juego de ping-pong (o tenis de mesa).

La ventaja es: es muy fácil aumentar para que funcione con más procesos, solamente hace falta usar más valores para `v` (es decir, la pelota se pasa explícitamente entre los participantes).

P246

Para visualizar este simple protocolo (también llamado *round robin*) lo implementamos en Java como un juego PingPong con árbitro.

Aquí está la clase principal, el árbitro.

Usamos la variable `turn` para que actue como la variable `v` del protocolo.

El árbitro lanza los dos jugadores y luego pone la pelota en el juego.

Observa, que aquí (todavía) no nos preocupamos de la terminación.

P247

La implementación directa del protocolo (observa los comentarios que coinciden con las líneas del pseudo-código).

La sección crítica es la salida a pantalla.

El cambio de la variable `turn` parece complicada, pero solamente varia el valor entre 1 y 2.

Implementa el juego en tu ordenador y notará que juegan infinitamente los dos siempre alternando ping con pong.

P248

Es un simple método para abreviar el uso de `sleep`, para no tener que usar el bloque `try-catch` en el código luego.

P249

Intentamos terminar el juego.

La idea es que el árbitro en un momento dado cambia el valor de la pelota (variable `turn`) a un valor diferente a 1 o 2, ya que eso provocaría que los jugadores salen de su bucle.

Observa que el árbitro espera 2 segundos antes de parar el juego.

Observa que ahora si hacemos la sincronización final como siempre con `join`.

Implementa esta modificación del hilo principal

Es importante que intentes realizar las tareas de programación ya que te quedará grabada una experencia de concurrencia para toda la vida.

P250

Es el mismo código del jugador para pensar como cambiar el funcionamiento para evitar el bucle infinito (línea con comentario `a:`).

¿Tienes una idea? recuerda que el árbitro pone `turn` a `3` para indicar a los jugadores que paren.

P251

Aquí está implementado una idea que se puede tener, seguramente se te ha ocurrido lo mismo:

- Cambiamos la condición del bucle externo para que juegue solamente mientras el árbitro no haya terminado.
- Cambiamos la condición del el bucle interno para que juegue solamente si es su turno y no haya terminado.

¿y ya está?

¿Qué observas cuando lo ejecutas? ¿funciona?

P252

Seguramente habrás observado que el código funciona de maravilla (yo por lo menos todavía no he observado que se quede parado).

¡Sin embargo: el programa es *incorrecto*!

Introduce nuestra espera (`Wait`) también en el jugador (se puede interpretar que lógicamente hacemos lo mismo solamento un poco más lento).

¡Es decir, el programa debe funciona igual!

Pero si ejecutas ahora, notarás que casi nunca funciona: siempre se queda bloqueado en algún momento.

¿sabes por qué pasa? ¿dónde está el fallo?

Parece que es un ejemplo simple pero refleja la situación que se observa muchas veces en el desarrollo de aplicaciones concurrentes: parecen que funcionan de maravilla hasta que se realiza algún cambio sencillo. Luego se pierde horas en averiguar qué estaba mal en el cambio, pero el problema no era el cambio, sino que el programa original estaba mal.

Como buena práctica para la depuración: en un programa concurrente correcto se puede introducir sentencias de demorra (nuestro `Wait`) en cualquier sitio y el programa debe funcionar igual.

P253

Retrocedemos al intento hacer un protocolo sin prioridad.

La idea de este segundo intento es usar para cada participante una variable que indica el acceso.

¿funciona? ¿qué puede ocurrir? (ayuda: piensa en un intento de acceso simultáneo...)

P254

Pues ya no hay alternancia, pero el protocolo es inseguro: si lo intentan los dos a la vez, ambos entran!

¡Observa bien cuando se da el caso de fallo!

P255

Nos puede ocurrir, cambiar el lugar donde modificacmos la variable de control (solamente hemos intercambiado líneas `b:` y `c:`.

¿funciona? ¿qué puede ocurrir? (ayuda: piensa otro vez en un intento de acceso simultáneo...)

P256

Está garantizado la exclusión mutua a (¡intenta comprobarlo con el teorema de la bandera!)

En este sentido el protocolo es seguro pero tenemos el problema que si los dos intentan entrar a la vez, ambos se quedan atrapados en el protocolo de entrada y no avanzan.

P257

En este cuarto intento aprovechamos de la idea de ceder el paso, tal como lo hemos visto en el protocolo asimétrico, pero claro, para que sea simétrico (sin prioridad en ningún lado), ambos tienen que ceder.

¿qué dices de esta solución?

P258

Pues la exclusión mutua sigue igual en pie.

Pero en casos muy sincronizados de ejecución se puede bloquear. Seguramente en la vida real (impleméntalo con Java en el pingpong si quieres) raras veces se puede observar un bloqueo durante mucho tiempo. (Seguramente solamente pasa cuando te toca ir en el avión que has programado tu :-)

P259

Pues el quinto intento también resuelve este problema del *livelock*.

Se aprovecha la idea del turno (primer intento) para garantizar alternancia en caso de intento de acceso simultáneo.

Con eso tenemos un protocolo para la exclusión mutua sin prioridad, es decir, una justicia *bastante justa*: quien quiere entrar puede entrar, si los dos lo intentan a la vez, se alterna.

P260

Bueno es todo para esta semana.