Material complementario 05/05/2020

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 vimos el uso de *locks* y *semaphores* para realizar secciones críticas. Hemos implementado de nuevo nuestro algoritmo de multiplicación con ellos y medir los resultados.
- También mejoramos el propio algoritmo para remediar el problema de la congestión sobre los mismos sitios (por eso se usa este ejemplo meramente didáctico), dividiendo el *trabajo* por realizar de tal manera entre los hilos que puedan actuar sin interferencia de los demás. Con eso vimos un aumento tremendo de la eficiencia, garantizando a su vez la corrección.
- Hemos visto de nuevo el uso de `wait` y `notify` (en Java) como ya también en las prácticas, para introducir el concepto abstacto de monitores para demostrar el uso de estos, por ejemplo, para liberar temporamente un cerrojo que será adquirido de nuevo de forma exclusiva despues de la notificación.

  Se ha resaltado tener el esquema de la máquina de estados de los hilos/procesos en consideración cuando se implementa aplicaciones concurrentes. Ojo, no todos estos conceptos son iguales en todos los entornos (en estas clases vemos solamente como está presente tales estados en Java).

- Se ha mencionado el concepto de *race condition* (condición de carrera), que también ya fue introducide en prácticas, que se manifesta en el problema que desde los requisitos de la aplicación se exige un orden temporal de ciertos eventos que hay que garantizar también si hay varios hilos/procesos diferentes gestionando tales acciones.

  Se quiere resaltar que a veces es complejo remediar un fallo en el diseño de una aplicación, si no se ha tenido en cuenta las posibles condiciones de carrera que se puedan producir en una aplicación. (Si quieres remedia el pingpong para que termine bien por aviso del árbitro.)

- Hemos visto diferentes intentos para llegar finalmente a un protocolo simetrico donde los dos hilos participantes tienen la misma prioridad para entrar en la sección crítica.

  Estos ejemplos también visualizan la complejidad que hay que tener en cuenta para conseguir tal tipo de sincronización entre los procesos.

  También se ha pedido que se intente verificar la corrección de los protocolos, si procede, usando el teorema de bandera.

P261

Son unos protocolos de la misma gama. Puede ser intersante su estudio en casos particulares en ciertas situaciones. Aquí pongo solamente referencias a información adicional para los interesados, nos basta con saber sus nombres por si acaso hay que buscarlos en el futuro.

Los algoritmos pueden ser interesante si se trabaja en un entorno distribuido sin memoria compartida, donde se tiene que implementar la exclusión mutua con el envio de mensajes.

P262

El hecho que se puede implementar protocolos de entrada y salida a secciones críticas meramento con `load` y `store` quizá ya resulta sorprendente en sí, pero se ha visto ya desde hace muchos años que no es eficiente.

Por eso ya desde los años 60 se ha implementado operaciones a nivel hardware que permiten la implementación de protocolos más simples y sobre todo más eficientes (tanto en número de instrucciones por ejecutar como en recursos en memoria necesarios).

Siguen siendo menos eficientes (ya que trabajan sobre memoria compartida) que las propias instrucciones que se ejecutan solamente dentro de la CPU.

Nombramos ahora unos de estas instrucciones hardware para tener una idea como funcionan y como se puede usar. También se mencionaran algunas de las ventajas que tienen comparado con las instrucciones simples de `load` y `store`.

P263

Lo importante es: la instrucción TAS es atómica, es decir, un proceso/hilo/nucleo/procesador (lo que sea) consigue que todas los pasos se lleven a cabo parecido en un solo *golpe* independientemente que hagan los demás (garantizado en hardware).

Tened en cuenta: puede ser que dentro de un sistema concreto, cuando varios procesos ejecutan una instrucción TAS al mismo tiempo, existe una arbitrariación con cierta justicia (por ejemplo que gana el nucleo con número más pequeño, o lo que sea). Si hace falta, por ejemplo para la implementación de sistemas operativos, hay que mirar las especificaciones hardware para averiguar como funciona en tal caso concretamento, sobre todo en sistemas críticos que tienen que garantizar requisitos de tiempo real.

P264

Observad lo siguiente: La variable `vi` no es necesaria se ha incluido aquí como indicador quien tenga acceso a la sección crítica. La línea `d:` es la instrucción TAS que se ejecuta de forma atómica.

Como se ve, un proceso queda atrapado en el bucle `c:d:e:` hasta que pueda entrar, y permite a otros la entrada en línea `i:`

Nota que no sabemos en este ejemplo, quien entra si varios procesos (¿ves que pueden ser muchos sin problema?) lo intentan a la vez. Por eso no podemos argumentar mucho sobre la justicia del protocolo sin conocer el hardware en concreto.

P265

Ya que no sabemos como se arbitra el orden como se ejecutan las operaciones en caso de intento simultaneo.

Para establecer un orden, la forma de realizar el protocolo es implementar un protocolo de paso, igual como lo hemos visto en prácticas. Puedes intentar hacerlo: un proceso manifesta en una variable que quiere entrar y va a espera con un TAS sobre otra variable (propio para él pero compartida con todos), un hilo que sale de la sección critica, elige según la justicia requerida el siguiente hilo que puede entrar poner su condición adecuadamente. Queda el problema como entrar si no hay nadie que queda por entrar.. ¿tienes una idea como tratar este caso?

P266

CAS es otra de las instrucciones atómicas a nivel hardware que se ha implementado para facilitar la concurrenia.

Está instrucción es algo *más potente* (tiene un número de consenso más alta, mira por ejemplo,) que la simple TAS, ya que se puede realizar el protocolo de dar paso de un proceso a otro de forma específica mucho más sencilla (pienso en guardar en la variable de condición `C` el hilo que debe entrar como siguiente...).

La operación CAS está disponible a nivel alto de Java en el paquete `java.util.concurrent.atomic`, más preciso, por ejemplo para la clase `AtomicReference` (Java 11): `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicReference.html`

P267

Hay muchas más operaciones disponibles cada una con su semántica y campo de aplicación. Podeis buscar en la Red con estás palabras para ver lo que hay, no entramos en más detalles aqui, no llega con entender el concepto principal y las dos, el TAS y el CAS.

Mirad por ejemplo el listado de operaciones disponibles para C++ moderno: `https://en.cppreference.com/w/cpp/atomic`

Como véis, se ha llevado estas instrucciones de bajo nivel directamente a alto nivel para que estén disponibles para su uso en la implementación de aplicaciones concurrentes. (Con el problema de posibles incompatibilidades si se quiere portar un programa de un sistema a otro.)

P268

He usado como ejemplo del compilador de gnu en su versión 8.3 (que raras veces se usa por defecto, seguramente en vuestra instalación de linux tenéis el 7.5.

Si echáis un vistazo al historial de las versiones del compilador véis que hay una gran cantidad de modificaciones y ampliaciones en los últimos años para dar mejor soporte al diseño e implementación de aplicaciones concurrentes.

P269

El problema llamada *ABA-problem* recoge el hecho que con las técnicas que hemos visto ahora, no se puede saber si entre de dos accesos del mismo proceso a una sección crítica otras han entrado o no.

Dicho problema no tiene que ver con un fallo, sino significa que hay que estar conciente que pueda ocurrir. Como ejemplo, mira tu cuenta bancaria, si ayer has visto 1000 euros y hoy ves 1000 euros, solamente sabes que has tenido 1000 euros y sigues teniendo 1000 euros, pero no sabes si te ha llegado la beca de 400 euros y has pagado el aquiler de 400 euros durante la noche. Si tal información es intersante en el caso concreto de una aplicación, pues hay que implementar tal funcionalidad con una bitácora (*log*) (y por eso los bancos tienen el modo *imprimir extracto de transacciones*).

P270

Entramos en otro tema importante de concurrencia (y que es un problema que no existe en la programación secuencial, menos que se trata de un efecto de falta de entrada o salida obstruida): el bloqueo.

Uso el concepto abstracto procesos y recurso (en Java podría ser hilos y objetos, sobre cuales se ejecuta bloques sincronizados).

Intuitivamente seguramente lo tenéis claro: si yo espero que vosotros hagáis algo, y vosotros esperáis que yo haga algo, nos quedaremos eternamente esperando.

Una visualización clásica es el problema de los filósofos donde por ejemplo hay dos platos y un cubierto (tenedor y cuchillo), si un filósofo tiene el tenedor y el otro le cuchillo, ninguno puede comer mientras el otro no suelta lo suyo, pero tampoco comen si ambos sueltan. Obviamente hay que establecer un mecanismo para gestionar esta situación.

P271

Esas son las cuatro condiciones que se tienen que cumplir para que se producza un bloqueo.

Nota importante, el internet y muchas otras fuentes usan como tercera condición *no se puede abortar el acceso exclusivo de un proceso a un recurso*, pero personalmente no me gusta como condición, ya que el hecho que NO se puede hacer algo no es una condición constructiva.

Tal como está formulada en la transparencia, abortar el acceso se puede ver como una opción de solución: ya si dos lo intentan, uno lo consigue, el otro no, pero este segundo podría abortar el primero y acceder al recurso. Otro posible solución podría ser aumentar las capacidades del recurso. Un ejemplo podría ser: procesos son los estudiantes, y recursos son los libros en una biblioteca. Si pido que dos (o más) cojan un libre (en exclusión mutua) y luego el quinto en la lista de referencias de este libre, puede dar se el caso que varios estudiantes queden bloqueados: el segundo libro que buscan ya está en mano de otro. Interrumpir sería quitar el libro a otro (pero cuidado, ambos pueden quitar!), aumentar capacidades, sería duplicar los libros en la biblioteca.

P272

Se describe que existen bloqueo parcial. Como dicho antes, segurmente no todos los estudiantes están bloquedas buscando su segundo libre, solamente aquellos que tienen que buscar un libro que otro tiene ya en la mano.

P273

Dado que los bloqueos pueden ocurrir, hay que pensar como solucionar el tema. Es bastante obvio que un bloqueo es un problema grave: la aplicación ya no hace nada (seguramente conocéis esta situación y os ha ocurrido una vez, que de repente el sistema (o sea el ordenador, el movil,...) yo no hace nada (y ojala tiene un botón para papagar...).

Veremos 3 posibilidades principales: la primera consiste en dejar que el bloqueo se produzca (es decir, se bloquean los procesos participantes) y se remedia, la segunda consiste en, durante la ejecución del programa, evitar que se produzca, y la tercera consiste en prevenir que se pueda producir un bloqueo ya por diseño de la aplicación.

P274

Miramos la primera posibilidad:

Dado que los procesos participantes se bloquean ellos mismos no pueden detectar que están bloqueados, por eso necesitamos un proceso adicional que *vigila* lo que está ocurriendo.

Se usa una estructura de datos llamado grafo que representa la situación actual de los procesos y sus accesos a los recursos. Se implementa cada petición con añadir una arista, cada acceso realizado con la invertir la arista, y cada liberación con eliminar la arista.

Como la cuarta condición dice que el bloqueo se produce cuando haya una cadena circular de peticiones y accesos, basta con analizar el grafo si contiene tal ciclo.

Observa que no hay que darse prisa, para decir lo de alguna manera, una vez formado el ciclo, no se deshace por si solo, ya que es un bloqueo.

El proceso adicional, cuando parece conveniente según requisitos de la aplicación, analiza el grafo para detectar ciclos, si no hay, pues la aplicación va bien, si detecta uno, tiene que actuar:

P275

pues, por ejemplo con las dos sugerencias mencionadas. Ya que si no permitimos (por lo menos durante algún tiempo, que un proceso acceso a un recurso con exclusión mutua, este queda libre para que otro proceso pueda conseguir acceso.

Depende de las características de las aplicaciones cual es la mejor solución.

Si retornamos al ejemplo con la biblioteca: tal proceso adicional puede ser el bibliotecario. Cuando ve que haya dos estudiantes cada uno con un libre ya adquerido, y ve que cada uno de ellos pide el libro del otro, pues puede actuar y pedir a uno de los dos que devuelva su libre, entonces lo asigna al compañero y cuando este termina con los dos, puede acceder el primero. Obviamente funciona parecido si hay más entidades involucrados.

P276

Una desventaje puede ser que se cae en un tipo de *livelock*, es decir, se cae en ciclos de abortar y re-pedir accesos, pero no se consigue un avance.

Otra posible desventaja en un entorno distribuido es: el algoritmo es centralizado; todos los procesos tienen que actuar sobre el mismo grafo, y el proceso adicional tiene que tener acceso a todos para poder actuar.

P277

La idea detrás de *evitar* es que los procesos piden acceso al sus recursos preguntando a una entidad (proceso) central (llamando banquero). Este proceso conoce la situación de de todos los procesos en tal momento de la ejecución y satisface las peticiones de tal manera que un bloqueo es imposible.

El logro del algoritmo de Dijkstra es que encuentra una solución si tal solución existe.

P278

La idea detrás de la prevención es diseñar la aplicación de tal manera que un bloqueo sea imposible.

Dado que tenemos las 4 condiciones necesarias basta con conseguir que una de ellas no se cumpla.

P279

La primera puede ser analizar bien si de verdad hace falta acceso con exclusión mutua (hay que analizar bien las características de la aplicación, muchas veces se notará que se puede eliminar los casos con acceso exclusivo).

La idea escrita aquí es implementar un sistema de gestión ante el recurso, si hay varias peticiones, dicho gestor (o demonio) maneja localmente los accesos (por ejemplo, mandar a espera a un proceso que tenga acceso, para darse a otro).

P280

La segunda consiste en pedir todos los recursos a la vez, o bien se consigue todos o bien ninguno. De esta manera un bloqueo es imposible.

Imagínate que aumantamos Java de tal manera que pueda hacer eso: sería por ejemplo permitir con el `synchronized` que se especifique varios objetos en vez de solo uno. Entonces, la máquina virtual de Java podría implementar esta forma de acceder a todos a la vez.

(No está hecho en Java, y una de las razones es que también hay que pensar en los `wait` y `notify` que complica la cosa...)

P281

Trabajar con la tercera condición es por ejemplo, quitar un recurso de otro proceso.

Observa que eso parece "bruto" pero en el fondo no lo es dado que el proceso del cual se quita el recurso no puede actuar ya que estamos, digamos, casi bloqueados (es decir, si no quitamos estaríamos bloqueados). El desafio de implementarlo en la realidad es no quitar lo que no se debe quitar.

P282

La cuarta condición se puede prevenir con acordar un orden como los procesos piden los recursos: es decir, ordenamos los recursos y exigimos que todos los recursos se piden respetando este orden si necesitamos más que uno.

De esta manera no se pueden formar ciclos.

La prevención es obviamente la mejor forma de garantizar que no haya bloqueos, pero no siempre se puede prevenir ya que depende de la aplicación en concreto (sobre todo si hay recursos que se crean dinámicamente o que dependen de entradas del programa).

Por eso se tiene que saber que existen las otras dos posibiliades.

P283

Pues un típico ejemplo que puede provocar un bloqueo en Java.