Sección 6. Condiciones de carrera.

definiciones

Si dos subprocesos(hilos) ejecutan n = n 1 en una variable compartida n aproximadamente al mismo tiempo, sus instrucciones de carga y almacenamiento podrían intercalar para que un hilo sobrescriba la actualización del otro.

Esta actualización perdida conduce a un resultado erróneo y es un ejemplo de una condición de carrera. Condiciones de carrera son posibles cuando dos o más subprocesos (hilos) comparten datos, están leyendo y escribiendo los datos compartidos al mismo tiempo, y el resultado finalde la computación depende de cuál hace qué y cuándo.

**Ejemplo de las condiciones de carrera**

**Ejemplo: race.java en la página 24** demuestra una actualización perdida en la que se reproduce = fn (suma, m) el papel de n = n + 1. La ejecución de muestra de race.java en la página 25 ilustra los resultados

**en el Ejemplo rac2.java en la página 25**, una condición de carrera entre un hilo ATM y un Auditor

Existe un hilo en un banco. Ejemplo de ejecución de rac2.java en la página 26 muestra los resultados.

**El ejemplo srbb.java en la página 27** muestra que no deberíamos sincronizar los hilos con suspend ()

y reanudar () porque una condición de carrera es posible. Si tratamos de reemplazar ocupado esperando con Bloqueo en el productor y consumidor de búfer limitado al tener un subproceso suspendido

hasta que sea reanudado por el otro hilo, corremos el riesgo tanto del hilo productor como del

Hilo del consumidor suspendido, cada uno esperando que el otro lo reanude.

**El controlador bbdr.java en la página 20** crea las hebras productor y consumidor. Ejecución de la muestra de srbb.java en la página 28 muestra una muestra una ejecución.

Los siguientes paneles muestran el código descrito en esta sección. Para ver el código, haga clic en

Siguiente; o puede ir directamente a la siguiente sección, Bloques sincronizados en la página 30, y regresara las muestras de código en otro momento.

**Sección 7. Bloques sincronizados.**

Objetos de bloqueo

Cada objeto de Java tiene un bloqueo. Un bloque sincronizado utiliza el bloqueo de un objeto para actuar como un binario.

semáforo con el valor inicial "1", resolviendo el problema de la sección crítica de exclusión mutua:

Es decir, todo el cuerpo del método de instancia es un bloque sincronizado en el objeto

(palabra clave esta) el método está en.

El JLS no garantiza que el hilo que ha esperado más tiempo para bloquear un objeto

Sé el siguiente en obtener el bloqueo cuando el objeto está desbloqueado.

**Ejemplos de bloques sincronizados.**

El ejemplo parp.java en la página 31 ofrece la generación de números primos multiproceso con un fijo

número de hilos (usando Class Prime.java en la página 7). Ejecución de muestra de parp.java en la página 31

muestra los resultados.

En el ejemplo norc.java en la página 32, solo se permite ejecutar un hilo a la vez

suma = fn (suma, m). Ejecución de muestra de norc.java en la página 32 muestra la ejecución de muestra.

Los siguientes paneles contienen un ejercicio y muestran el código descrito en esta sección.

Para ver el ejercicio y el código, haga clic en Siguiente; o puedes ir directamente a la siguiente sección,

Monitores en la página 33, y regresar a las muestras de código en otro momento.

**Probando el ejercicio-**

Use un bloque sincronizado para eliminar la condición de carrera en el Ejemplo rac2.java en la página 25.

**Sección 8. Monitores**

Monitorear la estructura y las propiedades.

Cada objeto Java posee un bloqueo y estos métodos: esperar (), notificar () y

notificar a todos ().

Un hilo que invoque un método sincronizado debe adquirir el bloqueo del objeto que contiene

El método antes de ejecutar el código del método. El hilo se bloquea si el objeto ya está

bloqueado

Un monitor tiene la siguiente estructura o patrón.

Un subproceso de Java se interrumpe cuando otro subproceso llama a su método interrupt (). Esta

call establece una bandera en el hilo interrumpido que esta última puede verificar periódicamente, permitiendo una

subproceso para indicar a otro subproceso que se detenga o devuelva los recursos asignados si no está en el medio

De alguna operación crítica. (Un hilo debe verificar su indicador de interrupción antes o después de tal

operaciones y tomar las medidas apropiadas cuando se interrumpa.

**Tenga en cuenta los siguientes puntos importantes:**

\* El hilo bloqueado por más tiempo en una llamada de método de monitor sincronizado no es

Garantizado para ser el siguiente hilo para adquirir el bloqueo del monitor cuando el bloqueo del monitor es

publicado.

\* No se garantiza que el subproceso bloqueado por más tiempo en una llamada de espera wait () sea el

una eliminada del conjunto de espera cuando hay una notificación () realizada por algún otro hilo en el

monitor.

\* La disciplina de señalización es señalizar y continuar, por lo que es posible hacer barcas - un hilo en espera

para que el bloqueo del monitor ejecute un método sincronizado del monitor podría obtener el bloqueo

antes de que un subproceso señalizado lo vuelva a adquirir, incluso si el notificar () ocurrió antes que el

Método de llamada de monitor. Así:

Example.

Cada objeto monitor tiene una única variable de condición anónima sin nombre. No podemos

señal con notificar () uno de varios hilos en espera de una condición específica. Es mas seguro

para utilizar NotifyAll () para despertar todos los hilos en espera para que puedan volver a verificar su espera

condiciones

\* Una notificación debe ser realizada por un hilo antes de esperar () si hay alguna variable de estado

fueron alterados por el hilo después de entrar en el monitor, lo que podría afectar a otros

Condiciones de espera de hilo. Esto también se aplica antes de abandonar el monitor (regresando de

el método).

\* Los campos de datos en un monitor no necesitan ser declarados volátiles porque todas las escrituras a

las variables compartidas por un hilo se completan antes de obtener y antes de liberar el

bloqueo del monitor.

\* Si un hilo que está bloqueado dentro de una llamada a suspensión (ms), join () o wait () es

interrumpido, entonces estos métodos borran la bandera de interrupción del hilo y lanzan un

Excepción interrumpida en lugar de regresar normalmente. Tenga en cuenta que ninguna excepción es

arrojarse si se interrumpe un hilo mientras está bloqueado esperando para adquirir un bloqueo de monitor para

Ejecutar un método sincronizado.

\* Por el contrario, InterruptedException se produce a la espera () si un subproceso que ha sido

la notificación se interrumpe mientras está bloqueada y en espera de volver a adquirir el bloqueo del monitor. Si una

InterruptedException ocurre cuando un hilo está en wait (), el hilo debe

vuelva a adquirir el bloqueo del monitor antes de ejecutar el código en el bloque catch.

\* Ignorando InterruptedException, como en:

mientras que (! condición) intente {esperar (); }

catch (InterruptedException e) {}

es indeseable El método de cierre debe devolver la excepción a la persona que llama.

\* El siguiente código:

si (! condición) intente {esperar (); }

catch (InterruptedException e) {}

es incorrecto porque un subproceso interrumpió su espera () y luego vuelve a entrar en el monitor

sin ser notificado.

\* Cuando se devuelve una llamada a esperar (milisegundos), el programa no puede determinar con seguridad si

se notificó la espera o si la espera se agotó después de transcurridos los milisegundos.

\* En algunas situaciones, podemos utilizar Notificar () en lugar de notificationAll () y si ...

espere () en lugar de mientras espera ... (). Sin embargo, es extremadamente complicado y no

Se recomienda debido a una condición de carrera entre interrumpir () y notificar ().

Supongamos que varios hilos están bloqueados dentro de wait () y luego uno de ellos es notificado

y luego se interrumpe antes de que vuelva a adquirir el bloqueo del monitor. El notificar () se "pierde" en

que uno de los otros hilos en espera debería continuar ahora. Necesitamos atrapar el

excepción cuando un hilo se interrumpe fuera de espera () y vuelve a generar la notificación ().

\* Por lo general, es incorrecto Thread.sleep (ms) mientras se encuentra dentro de un objeto monitor que contiene el

bloqueo (durante una invocación de método sincronizado). Otros hilos que quieran entrar en el

el monitor se bloqueará para adquirir el bloqueo del objeto del monitor y no se pueden interrumpir

de este estado.

Es mejor poner una bandera y dejar el monitor; otros hilos pueden esperar () a que

bandera para cambiar Ningún hilo retiene el bloqueo del monitor durante más tiempo que el establecido o el control

esta bandera

Como muestran los siguientes ejemplos, los monitores se pueden usar para sincronizar los hilos que solicitan

Los recursos de y devuelven recursos a un servidor. Esto se llama una relación cliente / servidor.

Los clientes interactúan con el servidor pero no entre ellos. El monitor del servidor es un pasivo.

Objeto en el sentido de que ningún hilo independiente se ejecuta dentro de él; el código en el monitor es

ejecutado solo cuando un subproceso de cliente invoca un método de monitoreo.

Los monitores pueden ser difíciles de usar si los subprocesos tienen una relación distinta de un cliente / servidor

uno.

Material de base sobre monitores.

Los semáforos son como gotos e indicadores: funcionan bien pero son propensos a errores y carecen de

Estructura y "disciplina".

Un monitor es un objeto con algunas funciones de exclusión mutua y sincronización de subprocesos.

capacidades Los monitores son una parte integral del lenguaje de programación por lo que el compilador puede

Genera el código correcto para implementar el monitor. Solo un hilo puede estar activo a la vez

en el monitor ("activo" significa ejecutar un método del monitor).

Los monitores también tienen variables de condición en las que un hilo puede esperar si las condiciones no son correctas

Para que continúe ejecutándose en el monitor. Algún otro hilo puede entrar en el monitor y

Tal vez cambie el estado del monitor. Si las condiciones son ahora correctas, ese hilo puede señalar una

hilo de espera, moviendo este último a la cola lista para volver al monitor cuando

se vuelve libre.

Los monitores pueden usar una disciplina de señalización de señal y salida o señal y continuar. En

señal y salida, un hilo de señalización debe abandonar el monitor inmediatamente, momento en el que se encuentra

Garantizado que el hilo señalado es el siguiente en el monitor.

En señal y continuar, no se garantiza que el hilo señalado sea el siguiente en el

monitor. De hecho, se puede realizar un barging - algunos hilos que han llamado un método de monitoreo y

está bloqueado hasta que el monitor esté libre, puede ingresar al monitor antes de un hilo señalado.

Se pueden usar semáforos y monitores para resolver la llamada sincronización "clásica"

problemas encontrados en muchos libros de sistemas operativos: el barbero dormido, los cinco comedores

Los filósofos, y los lectores y escritores de bases de datos.

El barbero dormido. Un peluquero espera para cortar el pelo. Los clientes entran en la sala de espera y toman

un asiento si hay uno disponible. Si la sala de espera está llena, vuelven a intentarlo más tarde. De lo contrario, esperan

Hasta su turno para un corte de pelo.

Cinco filósofos comedores. Cinco filósofos se sientan alrededor de una mesa y piensan hasta que tienen hambre.

Entre cada uno hay un tenedor (para un total de cinco tenedores). Para comer, un filósofo hambriento debe tener

Acceso exclusivo tanto a la horquilla a su izquierda como a la derecha. Si ambas horquillas no son libres, el

El filósofo espera.

El algoritmo del ejemplo dpmo.java en la página 39 no tiene interbloqueo (nunca sucede que todos

los filósofos tienen hambre, cada uno sosteniendo un tenedor y esperando el otro), permite máxima

paralelismo (un filósofo nunca levanta y sostiene un tenedor mientras espera que el otro tenedor

estar disponible cuando el tenedor que está sosteniendo podría ser utilizado para comer por su vecino), pero

también permite la inanición (los dos vecinos de un filósofo pueden colaborar y alternar sus

comiendo así el que está en el medio nunca puede usar los tenedores).

Si un filósofo puede sostener un tenedor mientras espera al otro tenedor, es posible un punto muerto, un

Caso extremo de no tener paralelismo máximo. Sin embargo, la inanición no es posible. Cada

la horquilla está representada por un semáforo y cada filósofo hambriento hace una "P" en su horquilla izquierda

y luego su horquilla derecha.

Podemos solucionar el problema del punto muerto y no retener la inanición, pero aún no tenemos el máximo.

paralelismo. Todos los filósofos recogen a la izquierda y luego a la derecha, excepto un filósofo designado que

recoge a la derecha y luego a la izquierda.

El hambre de los filósofos también se puede prevenir introduciendo un nuevo estado: muy hambriento. UNA

el filósofo es puesto en este estado si tiene hambre, si uno de sus vecinos baja los tenedores,

y si no puede comer porque el otro tenedor está en uso. Se agrega una nueva regla - un hambriento.

El filósofo no puede comer si tiene un vecino muy hambriento. Estos cambios impiden una

colaboración de dos filósofos que intentan matar de hambre al filósofo entre ellos.

Lectores y escritores. Se puede acceder a una base de datos de manera simultánea mediante hilos que solo desean

leer, pero un hilo de escritor debe tener acceso exclusivo con respecto a otros lectores y

escritores

Una solución podría permitir que los escritores se mueran de hambre si siguen llegando suficientes lectores para leer el

Base de datos para que el número de lectores actuales esté siempre por encima de cero.

El hambre de escritores se evita exigiendo a los lectores que vienen a leer la base de datos para

espere si hay un escritor en espera, incluso si otros lectores están leyendo la base de datos. Cuando

los lectores actuales terminan, el escritor en espera escribe la base de datos y luego señala en el

Base de datos de un lector en espera. Cada lector que entra señala a otro lector que espera en el

base de datos.

Ejemplos de monitores

El ejemplo dpmo.java en la página 39 ilustra el monitor de filósofos que comen. Cinco filósofos

Siéntate alrededor de una mesa y piensa hasta que tenga hambre. Entre cada pareja de filósofos hay un tenedor. UNA

El filósofo hambriento debe tener acceso simultáneo exclusivo a sus dos horquillas izquierda y derecha

para comer Si no son ambos libres, el filósofo espera. Driver dpdr.java en la página 40

Crea los hilos del filósofo. Ejecución de muestra de dpmo.java en la página 42 muestra la ejecución de muestra.

La siguiente figura ilustra el monitor de los filósofos comedores.

Imagen-----

**El ejemplo bbmo.java en la página 42** muestra un monitor de búfer acotado para un productor y

consumidor. Se manejan múltiples hilos de producción y múltiples hilos de consumo. Un productor

el hilo deposita elementos y bloques si el búfer acotado se llena. Un consumidor recupera hilos

elementos y bloques si el búfer acotado está vacío. El controlador bbdr.java en la página 20 crea el

Hilos de producción y consumo. Ejecución de muestra de bbmo.java en la página 43 muestra la ejecución de muestra.

La figura a continuación ilustra el monitor de búfer acotado para un productor y consumidor,

Manejo de múltiples hilos de producción y consumo.

**El ejemplo inmo.java en la página** 44 simula autos que se cruzan en una intersección de dos vías unidireccionales

calles para que:

\* Solo se puede cruzar un carro a la vez.

\* Un auto puede cruzar si no hay autos en la calle que cruza esperando para cruzar

\* Si dos autos se aproximan a la intersección al mismo tiempo, uno de ellos se cruzará (no

punto muerto)

\* Si hay autos en las calles que cruzan esperando, entonces los autos de la

Las calles que se cruzan para evitar el hambre.

Driver indr.java en la página 45 crea los subprocesos del coche. Ejecución de la muestra de inmo.java en la página 47

muestra los resultados.

La siguiente figura ilustra los autos que se cruzan en una intersección de dos calles de un solo sentido.

Los siguientes paneles contienen ejercicios y muestran el código descrito en esta sección. A

ver los ejercicios y el código, haga clic en Siguiente; o puedes ir directamente a la siguiente sección,

Semáforos en la página 50, y vuelva a las muestras de código en otro momento.

Prueba estos ejercicios

Ejercicio 1: escriba un monitor para el problema de los lectores y escritores de bases de datos. Lector múltiple

los hilos pueden leer la base de datos simultáneamente, pero los hilos de escritura deben tener exclusivos

Acceso con respecto a otros hilos de lectura y escritura.

Ejercicio 2: Escribe un monitor de barrera. Los hilos esperan hasta que todos los hilos lleguen a la barrera, luego

todos ellos son liberados

Ejercicio 3: cuando se realiza una notificación () o una notificación a todos () dentro de un monitor Java, el siguiente

El hilo para entrar en el monitor (adquirir el bloqueo) es arbitrario. Por lo tanto, los coches en el

la simulación de intersección que va en la misma dirección no necesariamente pasa por el

intersección en el orden en que llegaron a él (no es FCFS). Arregla esto.

**Sección 9. Semáforos.**

Clases escritas por el usuario

Las siguientes clases de semáforos escritas por el usuario son monitores de Java:

\* Abstract Class Semaphore.java en la página 53

\* Counting semaphore Class CountingSemaphore.java en la página 54

\* Semáforo binario Clase BinarySemaphore.java en la página 54

\* Clase de azúcar sintáctica SugarSM.java en la página 54 soP (S) se puede usar en lugar de S.P ().

362/5000

Material de base sobre semáforos.

Los semáforos se pueden utilizar para la exclusión mutua y la sincronización de subprocesos. En lugar de ocupado

esperando y desperdiciando ciclos de CPU, un hilo puede bloquearse en un semáforo (el sistema operativo

elimina el subproceso de la programación de la CPU o de la cola "listo") si debe esperar para ingresar su

Sección crítica o si el recurso que quiere no está disponible.

Escollos

El ejemplo de un semáforo de conteo que se muestra en la siguiente figura y en Clase

BadCountingSemaphore1.java en la página 54 no es correcto debido a ofbarging. Podemos reparar

El problema cambiando la semántica del campo de valor del semáforo. Para muestras de código,

vea Class Semaphore.java en la página 53, Class CountingSemaphore.java en la página 54, Class

BinarySemaphore.java en la página 54, Clase SugarSM.java en la página 54, Clase

BadCountingSemaphore1.java en la página 54, yClass BadCountingSemaphore2.java en

página 55.

Supongamos que el valor actual del semáforo es "0". Tres hilos invocan la p del semáforo ()

Método y espera. Luego, otro hilo llama a V (), que mueve uno de los tres en espera.

hilos al conjunto ejecutable.

Ahora, supongamos que un par de hilos adicionales avanzan por delante del hilo señalado y realizan dos

Más operaciones V () en el semáforo. Porque el valor del semáforo es positivo,

Notify () no se llama y ninguno de los subprocesos en espera se mueve al conjunto ejecutable.

Finalmente, el hilo señalado por la primera V () vuelve a entrar en el monitor semáforo, disminuye la

el valor del semáforo de "3" a "2", y abandona el monitor. El monitor tiene dos en espera.

Hilos y un valor positivo. Este es un estado inconsistente para un semáforo de conteo.

Para solucionar esto, podríamos intentar cambiar la semántica del campo de valor del semáforo, como en este

contando el semáforo, Class BadCountingSemaphore2.java en la página 55. El valor esta permitido

para ir negativo, en cuyo caso su valor absoluto es igual al número de subprocesos en espera.

Este enfoque corrige los problemas de barging pero introduce un problema de interrupt ().

Supongamos que el valor del semáforo es "-1" debido a un hilo bloqueado en wait () dentro de P (). Entonces

Supongamos que ese hilo está interrumpido. El valor se deja en "-1" aunque no haya subprocesos.

Bloqueado en P (). La próxima V () incrementará el valor a "0", mientras que ahora debería ser "1".

Otro problema más insidioso está presente: una condición de carrera entre interrupt () y

notificar(). Supongamos que varios hilos están bloqueados dentro de wait () y luego uno de ellos es

notificado y luego interrumpido antes de que vuelva a adquirir el bloqueo del monitor. El notificar () se "pierde"

en ese uno de los otros hilos en espera debe proceder ahora.

Por lo tanto, debemos detectar la excepción cuando se interrumpe un subproceso de espera () y

Regenera la notificacion ().

Ejemplos de semáforos

El ejemplo bbou.java en la página 56 muestra el productor de búfer acotado y consumer.Driver

bbdr.java en la página 20 crea las hebras productor y consumidor. Ejecución de la muestra de bbou.java

En la página 56 está la ejecución de la muestra.

El ejemplo dphi.java en la página 57 destaca a los filósofos del comedor. El conductor dpdr.java en la página 40

Crea los hilos del filósofo. La ejecución de la muestra es Ejecución de la muestra de dphi.java en la página 58.

Prueba estos ejercicios

Ejercicio 1: modifique el Ejemplo bbou.java en la página 56 para que maneje correctamente múltiples

Productor y múltiples hilos de consumo.

Ejercicio 2: escriba una solución de semáforo para el problema de los lectores y escritores de bases de datos.

Ejercicio 3: Escriba una solución de semáforo para los autos en un problema de intersección.

Problemas de seguridad en monitores y sincronización.

El método public void run () en un objeto Thread o Runnable puede ser invocado por cualquier

hilo que tiene una referencia al objeto Thread o Runnable. Así que evitamos que en el

comienzo del método run () con if (Thread.currentThread ()! = me) return ;.

El bloqueo de un objeto es accesible a cualquier hilo que tenga una referencia al objeto. Esto puede alterar

la operación de un monitor si algún hilo decide hacer algo como esto:

Sección 10. Mensaje que pasa.

Algunas definiciones

La programación orientada a objetos difumina la distinción entre invocar un método y enviar un

mensaje. También difumina la distinción entre computadora de memoria compartida y distribuida.

arquitecturas

La siguiente figura muestra la diferencia entre (1) invocar un método y (2) enviar un

mensaje:

1. El hilo deja el código en un objeto para ejecutar el código en otro objeto, luego regresa

(mostrado en la parte superior de la figura).

2. El hilo envía un objeto a otro hilo y luego, opcionalmente, bloquea hasta que el otro hilo

recibe el mensaje (que se muestra en la parte inferior de la figura).

El paso de mensajes lleva a una programación concurrente "más segura" ya que el objeto de recepción solo

tiene un hilo ejecutándose en su interior.

Es importante tener en cuenta que los subprocesos múltiples que invocan métodos en un objeto pueden llevar a la carrera

condiciones a menos que la sincronización se realice correctamente al hacer que el objeto sea un monitor. Con

paso del mensaje, el objeto receptor tiene un solo hilo ejecutándose dentro de él, lo que lleva a

Programación concurrente "más segura".

Si los hilos tienen una relación distinta de cliente / servidor, los monitores pueden ser difíciles de usar.

Usar el paso de mensajes entre los hilos es más fácil en estas situaciones. Si los hilos

Se comunican entre sí, se llaman pares o filtros. En esta situación, los hilos.

formar una tubería en la que cada subproceso recibe su entrada de su predecesor en la tubería y

envía su salida a su sucesor en la tubería.

Opciones para clases escritas por el usuario que implementan síncrono (bloqueo de envío) y

el paso de mensajes asíncronos (envío sin bloqueo) (recibir siempre bloques) incluye:

\* Envío de referencias de objetos de un hilo a otro dentro de la misma JVM

\* Envío de objetos serializados a través de sockets conectados desde un hilo en una JVM a una

hilo en otra JVM

Cada clase de paso de mensajes implementa un buzón o canal compartido por una colección de

trapos. El flujo unidireccional de información del remitente al receptor en un mensaje síncrono

pasar a veces se llama un simple encuentro. Los siguientes son ejemplos:

Material de referencia sobre el paso de mensajes

A veces, la frase "enviar un mensaje a un objeto" se usa para describir un hilo en uno

Objeto llamando a un método en otro objeto. Aquí, esa frase se usa para describir un hilo en

un objeto envía un mensaje a un hilo en otro objeto, donde el mensaje es en sí mismo una

objeto.

Esta técnica se utiliza para la comunicación de hilos y la sincronización en una computadora.

entorno donde los subprocesos no tienen memoria compartida (ya que los subprocesos residen en

diferentes máquinas virtuales o físicas). Por lo tanto, los hilos no pueden compartir semáforos o

Monitorea y no puede usar variables compartidas para comunicarse. El paso de mensajes todavía puede ser

Usado, por supuesto, en una plataforma de memoria compartida.

Los mensajes se envían a través de un puerto o canal con una operación como enviar (canal, mensaje)

y recibido desde un puerto o canal con una operación como recibir (canal, mensaje).

Los mensajes se pueden pasar sincrónicamente, lo que significa que el remitente se bloquea hasta que el receptor lo hace

una recepción y el receptor se bloquea hasta que el remitente realiza un envío.

Debido a que el remitente y el destinatario se encuentran en puntos conocidos específicos de su código en un conocido

En un instante específico de tiempo, el paso de mensajes sincrónicos también se denomina cita simple.

con un flujo unidireccional de información desde el remitente al receptor.

En el paso de los mensajes, el remitente no se bloquea. Si no hay receptor de heno.

a la espera de recibir el mensaje, el mensaje está en cola o en búfer. El receptor sigue bloqueando.

Si no hay un mensaje en cola o almacenado en búfer cuando se hace una recepción.

En el paso de los mensajes condicionales, el mensaje permanece en la cola hasta que se cumpla alguna condición,

Especificado por el receptor, se hace verdadero. En ese momento, el mensaje se pasa al receptor,

desbloquearlo.

Un flujo bidireccional de información, tal vez a través de la red, se denomina un encuentro extendido

y se puede implementar con un par de envíos y recepciones. Normalmente un hilo cliente usa esto

técnica para la comunicación con un hilo del servidor y solicita que se realice un servicio en su

favor.

Existe una situación similar cuando un subproceso de trabajo se pone en contacto con un subproceso maestro y solicita más

Trabajo por hacer. El cliente o el trabajador envía una solicitud y recibe la respuesta. El servidor o

El maestro recibe la solicitud, realiza el servicio y envía la respuesta.

El algoritmo Ejemplo qsrt.java en la página 65 se puede paralelizar para una memoria compartida,

máquina de CPU múltiple dedicando cada CPU a un subproceso de trabajo y usando un mensaje

Canal de paso como bolsa de tareas. El método main () pone toda la matriz a ordenar

en la bolsa

Un trabajador extrae la tarea, elige un punto de pivote y divide la matriz. Cada uno de los dos

Las particiones se vuelven a colocar en la bolsa como una tarea que debe realizar uno de los trabajadores.

A pesar de que el paso de mensajes se está utilizando para una serie de tareas, la memoria compartida sigue siendo

requerido porque la matriz se está ordenando "en su lugar" y las solicitudes de trabajo se están colocando en el

bag son pares de índices de matriz y no partes de la matriz en sí.

Un canal de comunicación de tareas, tarea de objeto, es compartido por el trabajador de ordenación rápida

trapos:

User-written clases (Clases escritas por el usuario)

Las siguientes son dos clases escritas por el usuario:

\* La clase SugarMP.java en la página 64 proporciona azúcar sintáctica para que se envíe (buzón, ms)

se puede usar en lugar de las instrucciones mail.send (ms) y mr = receive (buzón) en su lugar

de mr = buzón. recibe ().

\* Class MessagePassing.java en la página 64 envía referencias de objetos dentro de una JVM

asíncrono. El paso sincrónico de mensajes se puede hacer con el Rendezvous

clase, que discutiremos más adelante.

Ejemplos de paso de mensajes

El primer ejemplo contiene hilos de trabajo y una bolsa de tareas, el segundo ejemplo contiene

filtrar hilos, y el tercer ejemplo contiene hilos de pares.

\* Ejemplo de qsrt.java en la página 65: Quicksort (subprocesos de trabajo). Ejecute una muestra de qsrt.java en

La página 67 es la ejecución de muestra.

\* Ejemplo de pasv.java en la página 67: Tamiz paralelo de eratóstenes (hilos de filtro). Muestra

La ejecución de pasv.java en la página 69 es la ejecución de muestra.

\* Ejemplo de rads.java en la página 69: Parallel Radix Sort (hilos de pares). Ejecución de la muestra de

rads.java en la página 71 es la ejecución de muestra.

Puede usar la Invocación de método remoto (RMI) en la página 98 para implementar el paso de mensajes

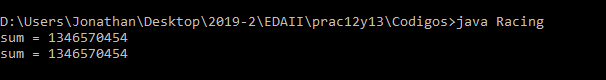
entre hilos en diferentes JVM que también pueden estar en diferentes máquinas físicas.

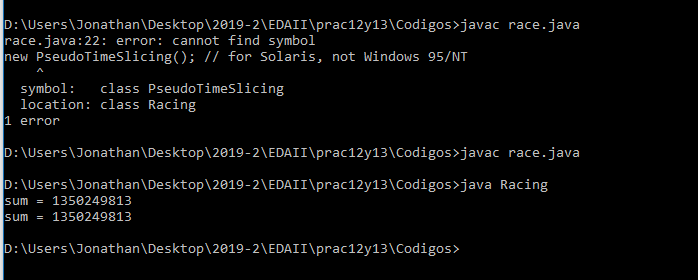
Los siguientes paneles muestran el código descrito en esta sección. Para ver el código, haga clic en

Siguiente; o puede ir directamente a la siguiente sección, Cita en la página 72, y regresar a la

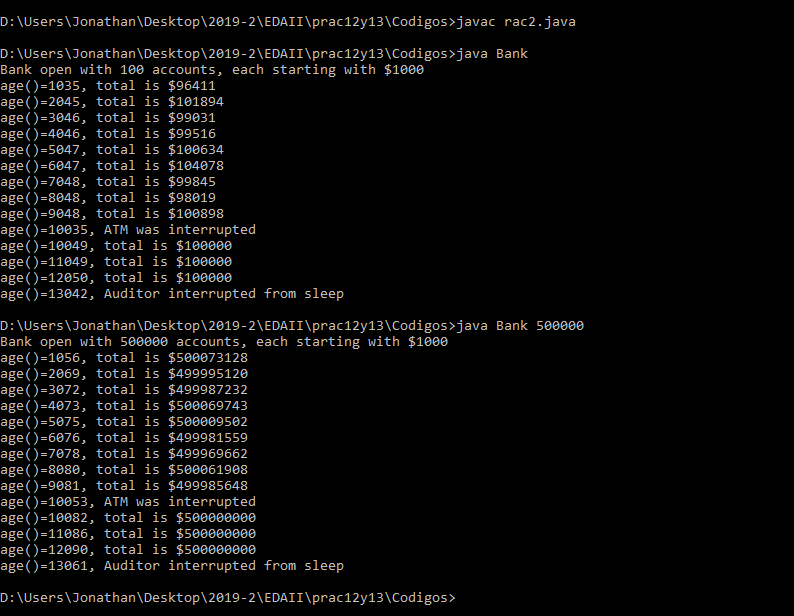
Muestras de código en otro momento.

EJEMPLO. 1

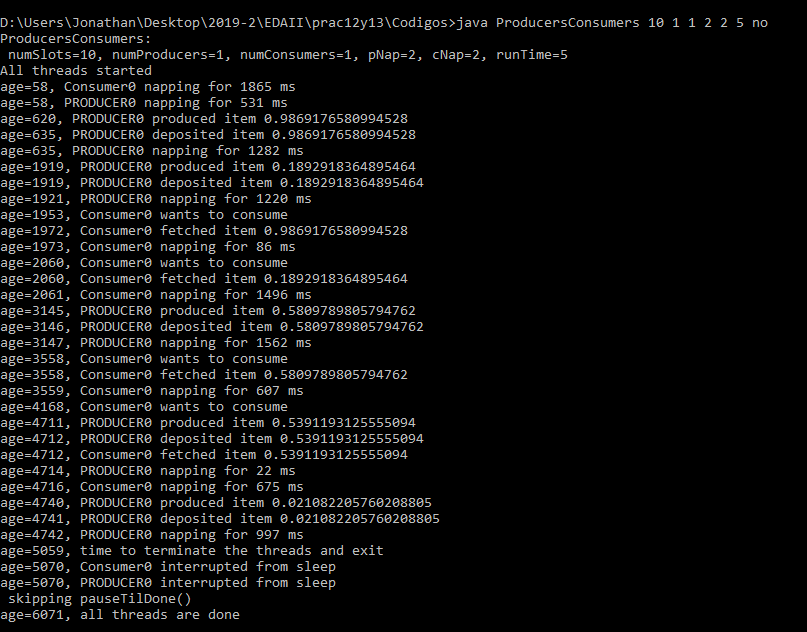




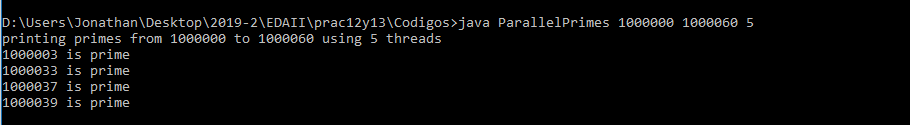
EJEMPLO: RAC2.JAVA



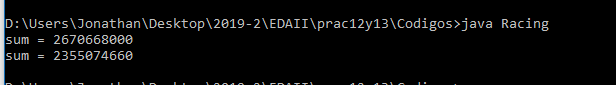
Sample run of srbb.java



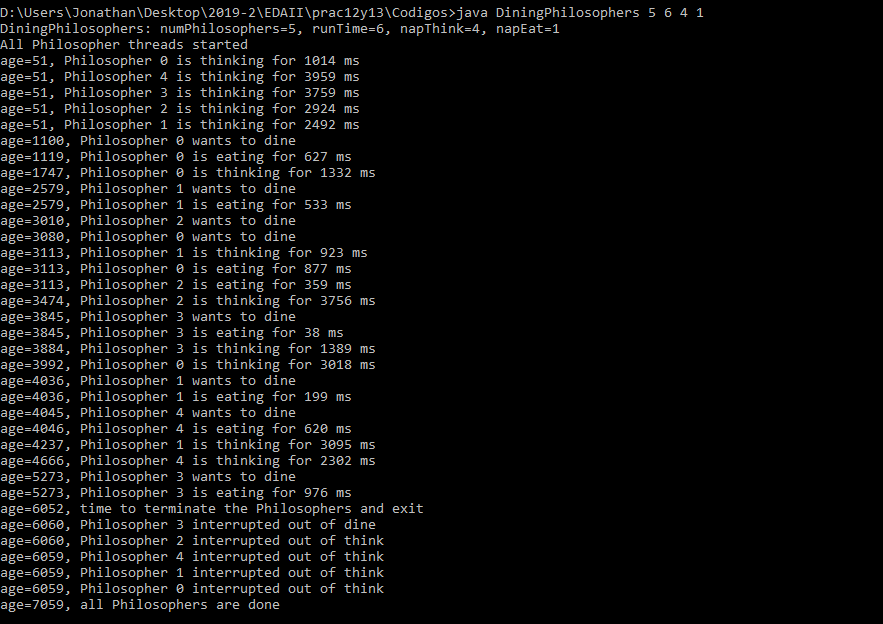
Corriendo ejemplo PARP.



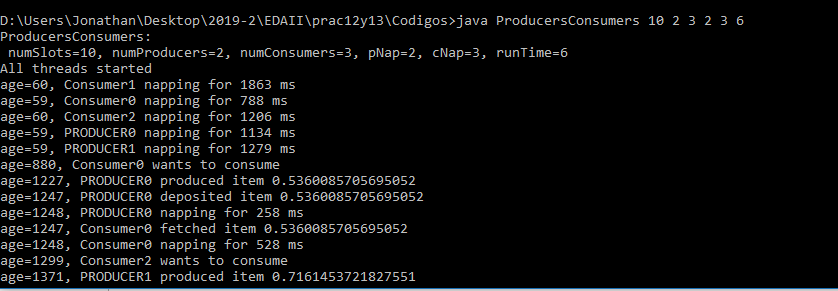
Corriendo norc.java

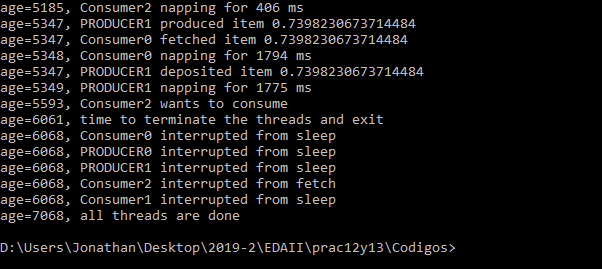


Corriendo dpmo.java

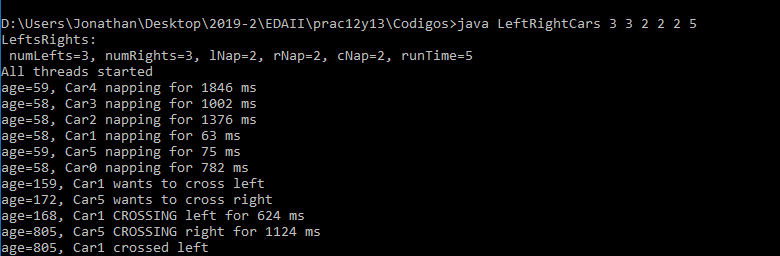


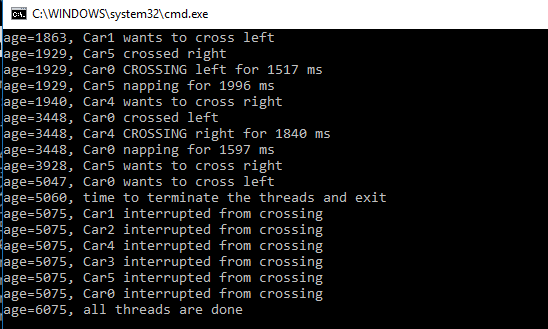
Corriendo bbmo.java



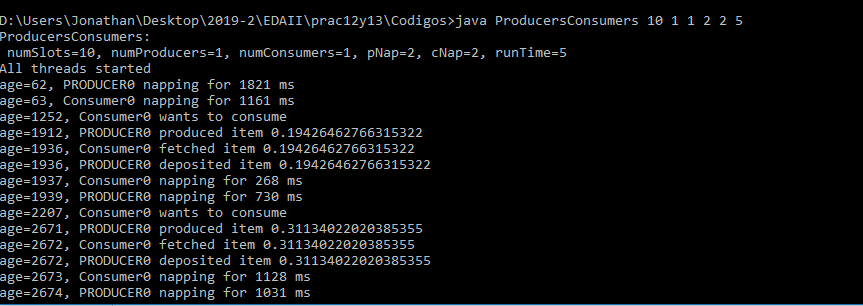


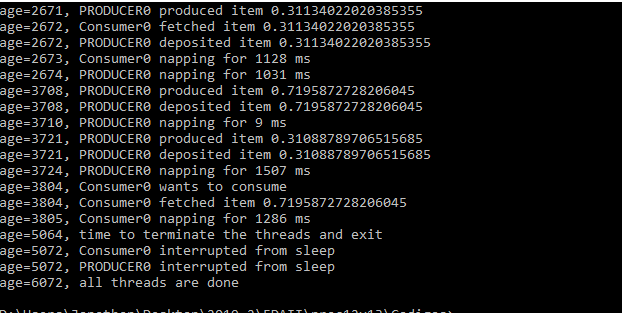
Corriendo Inmo.java



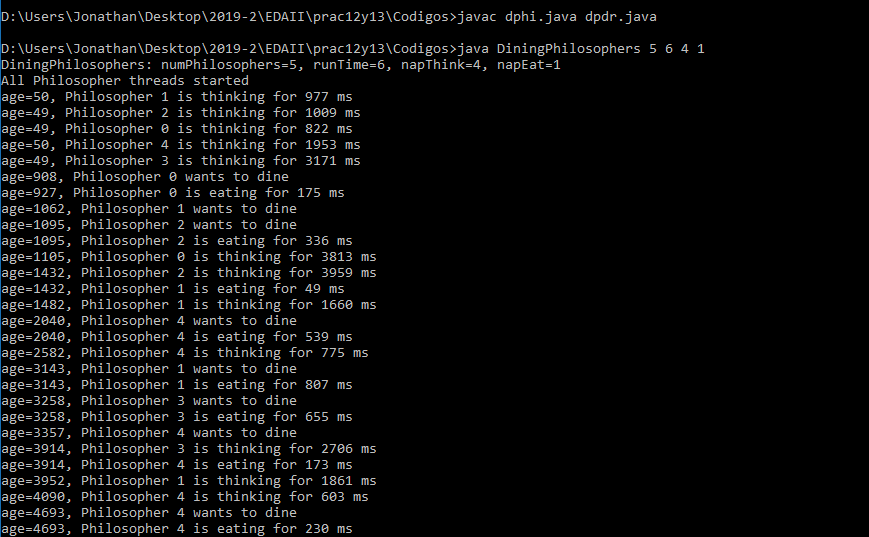


Corriendo bbou.java

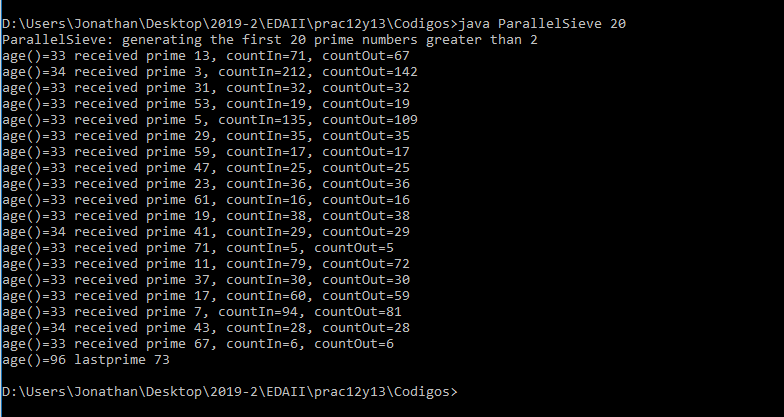




Corriendo dpi.java



Corriendo Pasv.java



Corriendo rads.java

