2019-2020

PSP -(P2)

JUAN RIVERA VAQUERO , 2ºDAM

1. Explica el concepto de región crítica:

Se denomina región crítica, a la porción de código de un programa en la que se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un proceso o hilo en ejecución. La sección crítica por lo general termina en un tiempo determinado y el hilo, proceso o tarea sólo tendrá que esperar un período determinado de tiempo para entrar.

1. ¿Qué es y cómo funciona una instrucción TAS (Test-And-Set)?

Es una instrucción hardware utilizada por ciertos procesadores para facilitar la creación de semáforos y otras herramientas necesarias para la programación concurrente en computadores.
Realiza dos acciones: leer el contenido de una palabra de la memoria en un registro y almacenar un valor distinto de cero en dicha palabra de memoria. Al tratarse de una instrucción máquina, el procesador nos garantiza que la instrucción TSL es realizada sin ningún tipo de interrupción por parte de otro proceso,

La instrucción test-and-set (TAS) implementa una comprobación a cero del contenido de una variable en la memoria al mismo tiempo que varía su contenido en caso que la comprobación se realizó con el resultado verdadero.

Así se puede realizar la exclusión mutua con

```
Initially:    vi is equal false
              C  is equal true

a:        loop
b:          non-critical section
c:          loop
d:              if C equals true          ; atomic test-and-set
e:                  set C to false and exit
f:        endloop
g:         set vi to true
h:        critical section
i:        set vi to false
j:         set C to true
h:   endloop
```

En caso de un sistema multi-procesador hay que tener cuidado que la operación test-and-set esté realizada en la memoria compartida.

Teniendo solamente una variable para la sincronización de varios procesos el algoritmo arriba no garantiza una espera limitada de todos los procesos participando. ¿Por qué?

Para conseguir una espera limitada se implementa un protocolo de paso de tal manera que un proceso saliendo de su sección crítica da de forma explícita paso a un proceso esperando (en caso que tal proceso exista).

2. Explica el Principio de la Bandera para comprobar la exclusión mutua. ¿Qué es? ¿Cómo funciona?

El principio de la bandera es un teorema que nos permite poder comprobar la propiedad de exclusión mutua en un código determinado. Dicho principio consiste en que antes de que dos procesos puedan acceder a una región crítica, deben levantar una bandera (poner a TRUE una variable boolean, por ejemplo) y después mirar a la bandera del otro, de esta forma por lo menos uno de los dos verá la bandera del otro, al ver la bandera del otro, tendrá que bajar su bandera y esperar a entrar a la región crítica a que la bandera del otro esté bajada. Hay que tener en cuenta que el principio de la bandera puede usarse para más de dos procesos, aunque esto complica la lógica del programa y sería más difícil comprobar el correcto funcionamiento. Para comprobar dicho principio, podemos utilizar la comprobación por contradicción que consiste en: 1. Asumimos que P0 era el último en mirar. 2. Entonces la bandera de P0 está levantada. 3. Asumimos que P0 no ha visto la bandera de P1. 4. Entonces P1 ha levantado la bandera después de que mirara P0. 5. Pero P1 mira después de levantar la bandera. 6. Entonces P0 no es el último en mirar. Se utiliza para comprobar que un sólo un proceso pueda obtener acceso a la sección crítica (garantía del acceso con exclusión mutua). Por lo menos un proceso debe obtener acceso a la sección crítica después de un tiempo de espera finito. Obviamente se asume que ningún proceso ocupa la sección crítica durante un tiempo infinito

3. ¿Qué es la Espera Activa en programación concurrente?

El algoritmo de Dekker y sus derivados provocan una espera activa de los procesos cuando quieren acceder a un recurso compartido. Mientras están esperando a entrar en su región crítica no hacen nada más que comprobar el estado de alguna variable. Normalmente no es aceptable que los procesos permanezcan en estos bucles de espera activa porque se está gastando potencia del procesador inútilmente. Un método mejor consiste en suspender el trabajo del proceso y reanudar el trabajo cuando la condición necesaria se haya cumplido. Naturalmente dichas técnicas de control son más complejas en su implementación que la simple espera activa. Ejemplo: un sistema FIFO con prioridades de los procesos a la hora de acceder a un recurso para evitar la espera activa.

4. ¿Qué es la Condición de Carrera en programación concurrente?

Es una expresión usada en programación. Cuando la salida o estado de un proceso es dependiente de una secuencia de eventos que se ejecutan en orden arbitrario y van a trabajar sobre un mismo recurso compartido, se puede producir un bug cuando dichos eventos no se ejecutan en el orden que el programador esperaba. El término se origina por la similitud de dos procesos «compitiendo» en carrera por llegar antes que el otro, de manera que el estado y la salida del sistema dependerá de cuál «llegó» antes, pudiendo provocarse inconsistencias y comportamientos impredecibles y no compatibles con un sistema determinista.

5. ¿Para qué sirve el modificador "volatile" en Java?

Se usa este modificador sobre los atributos de los objetos para indicar al compilador que es posible que dicho atributo vaya a ser modificado por varios threads de forma simultánea, asíncrona y que por tanto no queremos guardar una copia local del valor para cada thread (a modo de caché). Queremos que los valores de todos los threads estén sincronizados (a pesar de afectar al rendimiento). Volatile es más simple y sencillo que synchronized es decir, proporciona un rendimiento mejor. Sin embargo, no proporciona atomicidad por lo que debemos tener cuidado ya que solo garantiza visibilidad entre threads.

6. ¿Qué es un Monitor en programación concurrente? ¿Para qué sirve?

Los monitores son estructuras de datos abstractas destinadas a ser usadas sin peligro por más de un hilo de ejecución. La característica que principalmente los define es que sus métodos son ejecutados con exclusión mutua. Lo que significa, que en cada momento en el tiempo, un hilo como máximo puede estar ejecutando cualquiera de sus métodos. Esta exclusión mutua simplifica el razonamiento de implementar monitores en lugar de código a ser ejecutado en paralelo.

7. El paquete "java.util.concurrent" proporciona clases con herramientas útiles para la programación concurrente. Define para qué sirven las siguientes:

Clase Future:

La clase Future representa un resultado futuro de un cálculo asincrónico, un resultado que finalmente aparecerá en el Futuro después de que se complete el procesamiento.
Algunos ejemplos de operaciones que aprovecharían la naturaleza asíncrona de Future son:

- Procesos computacionales intensivos (cálculos matemáticos y científicos)
- Manipular grandes estructuras de datos (big data)
- Llamadas a métodos remotos (descarga de archivos, desguace de HTML, servicios web).

Clase CountDownLatch:

CountDownLatch se usa para asegurarse de que una tarea espere otros subprocesos antes de comenzar. Para comprender su aplicación, consideremos un servidor donde la tarea principal solo puede iniciarse cuando se hayan iniciado todos los servicios requeridos.

Funcionamiento de CountDownLatch:
cuando creamos un objeto de CountDownLatch, especificamos el número de subprocesos que debe esperar, todos esos subprocesos deben realizar una cuenta regresiva llamando a

CountDownLatch.countDown () una vez que estén completos o listos para el trabajo. Tan pronto como el recuento llega a cero, la tarea de espera comienza a ejecutarse.

Clase ExecutorService:

ExecutorService es un marco proporcionado por el JDK que simplifica la ejecución de tareas en modo asíncrono. En términos generales, ExecutorService proporciona automáticamente un conjunto de hilos y API para asignarle tareas.

Clase Semaphore:

Conceptualmente, un semáforo mantiene un conjunto de permisos. Cada adquirir () bloquea si es necesario hasta que haya un permiso disponible, y luego lo toma. Cada lanzamiento () agrega un permiso, potencialmente liberando a un adquirente de bloqueo. Sin embargo, no se utilizan objetos de permiso reales; el semáforo solo mantiene un recuento del número disponible y actúa en consecuencia.

Clase BlockingQueue:

Podemos distinguir dos tipos de BlockingQueue :

- cola ilimitada: puede crecer casi indefinidamente
- cola delimitada - con capacidad máxima definida

Cola ilimitada:

Crear colas ilimitadas es simple:

```
BlockingQueue<String> blockingQueue = new
LinkedBlockingDeque<>();
```

La capacidad de *blockQueue* se establecerá en *Integer.MAX_VALUE.* Todas las operaciones que agregan un elemento a la cola ilimitada nunca se bloquearán, por lo que podría crecer hasta un tamaño muy grande.

Lo más importante cuando se diseña un programa productor-consumidor utilizando BlockingQueue ilimitado es que los consumidores deberían poder consumir mensajes tan rápido como los productores agregan mensajes a la cola. De lo contrario, la memoria podría llenarse y obtendríamos una excepción *OutOfMemory* .

Cola limitada:

El segundo tipo de colas es la cola delimitada. Podemos crear tales colas pasando la capacidad como argumento a un constructor:

```
BlockingQueue<String> blockingQueue = new
LinkedBlockingDeque<>(10);
```

Aquí tenemos un *BlockQueue* que tiene una capacidad igual a 10. Significa que cuando un consumidor intenta agregar un elemento a una cola ya llena, dependiendo del método que se utilizó para agregarlo (*oferta ()* , *agregar ()* o *poner ()*), se bloqueará hasta que haya espacio disponible para insertar objetos. De lo contrario, las operaciones fallarán.

El uso de la cola delimitada es una buena manera de diseñar programas concurrentes porque cuando insertamos un elemento en una cola que ya está llena, esas operaciones deben esperar hasta que los consumidores se pongan al día y hagan espacio disponible en la cola. Nos da aceleración sin ningún esfuerzo de nuestra parte.

Clase AtomicInteger:

La clase AtomicInteger protege un `int` valor subyacente al proporcionar métodos que realizan operaciones atómicas en el valor. No se utilizará como reemplazo de una `Integer` clase.

Es una clase de contenedor para un int valor que permite que se actualice atómicamente. La clase proporciona métodos útiles, el uso más común de este AtomicInteger es manejar un contador al que acceden diferentes hilos simultáneamente.

En usos de la vida real, necesitaremos `AtomicInteger` en dos casos:

1. Como un contador atómico que está siendo utilizado por múltiples hilos simultáneamente.
2. En operaciones de comparación e intercambio para implementar algoritmos sin bloqueo.