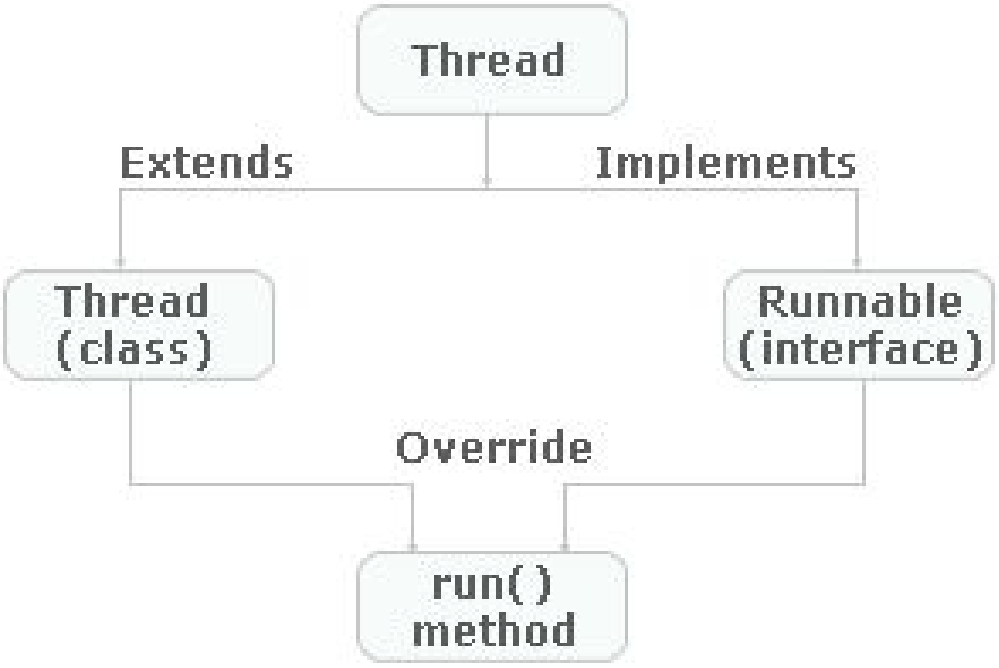
Prácticas 2



Pablo Ramilo Costal

1.-

En programación concurrente se denomina sección crítica o región crítica a la porción de código de un programa de ordenador el cual no debe de ser accedido por más de un hilo en ejecución (thread) al mismo tiempo, determina una forma de prohibir que más de un proceso lea o escriba en los datos compartidos en un mismo tiempo. La sección crítica por lo general termina en un tiempo determinado y el hilo, proceso o tarea solo tendrá que esperar un periodo determinado de tiempo para entrar. Se necesita de un mecanismo de sincronización en la entrada y salida de la sección crítica para asegurar la utilización exclusiva del recurso, por ejemplo un semáforo.

La sección crítica se utiliza por lo general cuando un programa multihilo actualiza múltiples variables sin un hilo de ejecución separado que lleve los cambios conflictivos a esos datos.

La manera en cómo se implementan las secciones puede variar dependiendo de los diversos sistemas operativos.

Sólo un proceso puede estar en una sección crítica a la vez.

.

El método más común para evitar que dos procesos accedan al mismo tiempo a un recurso es el de la exclusión mutua, donde se inhabilita las interrupciones durante el conjunto de instrucciones más pequeño para que no se corrompa el recurso.

Synchronized es un modificador en java que permite controlar la entrada de los hilos en la sección crítica.

2.-

El TAS (test and set) es una instrucción, que no puede ser interrumpida, de hardware que 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, si es 0 lo pone a 1 y cuando haya acabado cambia el estado del recurso compartido para permitir la entrada a otro repitiendo así este proceso.

3.-

Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control que permite la entrada de un proceso. Debido a esto dekker creó un sistema donde dos hilos quieren entrar a una sección crítica, los dos levantan su bandera y miran en busca de otro más que la tenga levantada hasta quedar uno con la bandera levantada y se ejecuta mientras que el resto espera con la bandera bajada. Al terminar el primero vuelve a levantar su bandera para decir que ha terminado y que puede entrar el siguiente. Debido a que esto funciona bien en dos hilos pero con más no tiene el mismo resultado, se crearon más algoritmos a partir de este, por ejemplo el de la panadería de Lamport.

La panadería de Lamport toma su nombre de la costumbre en panaderías y tiendas en general donde las personas toman un número único y esperan su turno. En el algoritmo al entrar un hilo que se le asigna el número máximo +1 y lo da el turno en false y los compara con todos los hilos para saber si hay alguno antes de ese, y espera a que se termine de ejecutar los anteriores para cambiar a 0.

4.-

Se denomina espera activa o espera ocupada a una técnica donde un proceso verifica una condición constantemente, como la espera de una entrada por teclado o el ingreso habilitado a una sección crítica, y puede ser válida en circunstancias especiales como en la sincronización de procesos en los sistemas con múltiples procesadores (SMP) pero debe ser evitada ya que consume recursos del cpu sin realizar ninguna operación.

Un método mejor consiste en suspender el trabajo del proceso y reanudar el trabajo cuando la condición necesaria se haya cumplido, aunque estas técnicas de control son más complejas de implementar.

5.-

La condición de carrera (race condition) ocurre cuando dos o más procesos acceden un recurso compartido sin control, de manera que el resultado combinado de este acceso depende del orden de llegada, de manera que se puede producir un error cuando dichos eventos no *llegan* (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.

Se pueden dar condiciones de carrera a nivel de proceso o incluso a nivel de sistema cuando este está distribuido.

A modo de ejemplos de alto nivel: dos personas ingresando y retirando efectivo a la vez de una misma cuenta bancaria podrían ver su saldo incrementado o por el contrario su ingreso realizado pero no materializado en saldo disponible o un sistema mal diseñado de reserva de entradas podría ocasionar que dos usuarios accediendo desde la web a la vez, reserven legítimamente la misma butaca. Si bien son situaciones que pudieran ser improbables, son posibles y pueden ser y han de ser evitadas en el desarrollo de software.

La naturaleza imprevisible de las condiciones de carrera da lugar en muchas ocasiones a la aparición de bugs de manera repentina que normalmente no ocurren durante el testeo de un software.

En estos casos se deberían tomar las precauciones necesarias para que las acciones llevadas a cabo por cada recurso fuesen atómicas, sin embargo, hay ocasiones en las que no los son, ocasionando datos inconsistentes o dejando en

ocasiones abierto un intervalo de tiempo en el que un atacante podría obtener privilegios, leer y escribir sobre ficheros protegidos.

6.-

volatile es más simple y más sencillo que synchronized, lo que implica también un mejor rendimiento. Sin embargo, volatile, a diferencia de synchronized, no proporciona atomicidad (que o se ejecuten todos los pasos o ninguno), lo que puede hacer que sea más complicado de utilizar.
Una operación como el incremento, por ejemplo, no es atómica. El operador de incremento se divide en realidad en 3 instrucciones distintas (primero se lee la variable, después se incrementa, y por último se actualiza el valor) por lo que algo como lo siguiente podría causarnos problemas a pesar de que la variable sea volatile:

Al declarar una variable como volátil habrá una variable para cada objeto. Así que en la en la programación no hay ninguna diferencia con respecto a una variable normal, y totalmente diferente de estática. Sin embargo, incluso con los campos de objeto, un hilo puede almacenar en memoria caché un valor de la variable locamente.
Esto significa que si dos subprocesos actualizar una variable del mismo objeto al mismo tiempo, y la variable no es declarada volátil, podría ser un caso en el que uno de los hilos tiene en caché un valor anterior.
Incluso si tiene acceso a un archivo (clase) de valores estáticos a través de hilos múltiples, cada hilo puede tener su copia en caché. Para evitar esto, puede declarar la variable como volátil estática y esto hará que el hilo tenga que leer nuevamente la variable cada vez asegurando que se tendrá el valor más actual.

7.-

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 aunque muchos esten llamando al método. Esta exclusión mutua simplifica el razonamiento de implementar monitores en lugar de código a ser ejecutado en paralelo.

El problema de que dos threads ejecuten un mismo procedimiento dentro del monitor es que se pueden dar condiciones de carrera, es decir, que compitan por entrar y el mas rapido entra, perjudicando el resultado de los cálculos. Para evitar esto y garantizar la integridad de los datos privados, el monitor hace cumplir la exclusión mutua implícitamente, de modo que sólo un procedimiento esté siendo ejecutado a la vez. De esta forma, si un thread llama a un procedimiento mientras otro thread está dentro del monitor, se bloqueará y esperará en la cola de entrada hasta que el monitor quede nuevamente libre. Aunque se la llama cola de entrada, no debería suponerse ninguna política de encolado.

Los monitores deben incluir algún tipo de forma de sincronización como por ejemplo una condición como mecanismo de bloqueo del thread, a la vez que se debe liberar el monitor para ser usado por otro hilo. Más tarde, cuando la condición permita al thread bloqueado continuar ejecutando, debe poder ingresar en el monitor en el mismo lugar donde fue suspendido. Para esto los monitores poseen variables de condición que son accesibles sólo desde adentro. Existen dos funciones para operar con las variables de condición:

- cond_wait(c): suspende la ejecución del proceso que la llama con la condición *c*. El monitor se convierte en el dueño del lock y queda disponible para que otro proceso pueda entrar.
- cond_signal(c): reanuda la ejecución de algún proceso suspendido con *cond_wait* bajo la misma condición *c*. Si hay varios procesos con esas características elige uno. Si no hay ninguno, no hace nada.

8.-

-Clase Future: representa el resultado futuro de un cálculo asíncrono, cuando una tarea asíncrona es creada esta devuelve un objeto future y este objeto Future funciona como un identificador del resultado de la tarea asincrónica. Una vez se complete la tarea se puede acceder al resultado a través del objeto Future devuelto cuando se inició la tarea.

La clase Future también es usada en otras clases como en algunos métodos de ExecutorService que devuelve un objeto future cuando envía un Callable para que se ejecute simultáneamente (asincrónicamente).

ejemplo próximo a la definición:

```
public interface Future<V> {
   boolean cancel(boolean mayInterruptIfRunning)
   V       get();
   V       get(long timeout, TimeUnit unit);
   boolean isCancelled();
   boolean isDone();
}

Future future = ... // Get Future from somewhere

if(future.isDone()) {
   Object result = future.get();
} else {
   // do something else
}
```

-Clase CountDownLatch:
Permite que uno o más subprocesos esperen hasta que se complete un conjunto de operaciones que se realizan en otros subprocesos.
Un CountDownLatch se inicializa con un recuento determinado. Los métodos de espera se bloquean hasta que el recuento actual llega a cero debido a las Callables del método countDown (), después de lo cual se liberan todos los subprocesos en espera y cualquier invocación posterior de espera regresa de inmediato.

Un CountDownLatch inicializado con un recuento de uno sirve como un simple pestillo de encendido / apagado, o puerta: todos los hilos que invocan esperan en la puerta hasta que se abre un hilo que invoca countDown (). Un CountDownLatch inicializado en N puede usarse para hacer que un hilo espere hasta que N hilos hayan completado alguna acción, o alguna acción se haya completado N veces.

Una propiedad útil de un CountDownLatch es que no requiere que los subprocesos que llaman a countDown esperen a que el recuento llegue a cero antes de continuar, simplemente evita que cualquier subproceso pase de una espera hasta que todos los subprocesos puedan pasar.

declaración en una clase runnable:

```
public class Worker implements Runnable {
    private List<String> outputScraper;
    private CountDownLatch countDownLatch;

    public Worker(List<String> outputScraper, CountDownLatch countDownLatch) {
        this.outputScraper = outputScraper;
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run() {
        doSomeWork();
        outputScraper.add("Counted down");
        countDownLatch.countDown();
    }
}
```

test:

```
public void whenParallelProcessing_thenMainThreadWillBlockUntilCompletion()
  throws InterruptedException {

    List<String> outputScraper = Collections.synchronizedList(new ArrayList<>());
    CountDownLatch countDownLatch = new CountDownLatch(5);
    List<Thread> workers = Stream
      .generate(() -> new Thread(new Worker(outputScraper, countDownLatch)))
      .limit(5)
      .collect(toList());

      workers.forEach(Thread::start);
      countDownLatch.await();
      outputScraper.add("Latch released");

      assertThat(outputScraper)
        .containsExactly(
          "Counted down",
          "Counted down",
          "Counted down",
          "Counted down",
          "Counted down",
          "Latch released"
        );
    }
```

-Clase ExecutorService:

Proporciona métodos para gestionar la terminación y métodos que pueden producir un futuro para seguir el progreso de una o más tareas asincrónicas.
Un ExecutorService puede cerrarse, lo que hará que rechace nuevas tareas. Se proporcionan dos métodos diferentes para cerrar un ExecutorService. El método shutdown () permitirá que las tareas enviadas previamente se ejecuten antes de

finalizar, mientras que el método shutdownNow () evita que las tareas en espera comiencen e intenta detener la ejecución de las tareas actuales. Al finalizar, un ejecutor no tiene tareas ejecutándose activamente, no hay tareas en espera de ejecución y no se pueden enviar nuevas tareas. Un ExecutorService no utilizado debe cerrarse para permitir la recuperación de sus recursos.

El envío de método extiende el método base Executor.execute (java.lang.Runnable) creando y devolviendo un futuro que se puede usar para cancelar la ejecución y / o esperar a que se complete. Los métodos invokeAny e invokeAll realizan las formas más útiles de ejecución masiva, ejecutan una colección de tareas y luego esperan que se complete al menos una o todas. (Class ExecutorCompletionService se puede usar para escribir variantes personalizadas de estos métodos).

La clase Executors proporciona métodos de fábrica para los servicios de ejecutor proporcionados en este paquete.

```
ExecutorService executorService =
  new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
  new LinkedBlockingQueue<Runnable>());

Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};
```

```
List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(callableTask);
callableTasks.add(callableTask);
callableTasks.add(callableTask);
```

-Clase semaphore:

Un semáforo contando. 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.
Los semáforos a menudo se usan para restringir el número de hilos que pueden acceder a algún recurso (físico o lógico).

```
class Pool {
  private static final int MAX_AVAILABLE = 100;
  private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

  public Object getItem() throws InterruptedException {
    available.acquire();
    return getNextAvailableItem();
  }

  public void putItem(Object x) {
    if (markAsUnused(x))
      available.release();
  }

  protected Object[] items = ... whatever kinds of items being managed
```

```
  protected boolean[] used = new boolean[MAX_AVAILABLE];

  protected synchronized Object getNextAvailableItem() {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (!used[i]) {
        used[i] = true;
        return items[i];
      }
    }
    return null; // not reached
  }

  protected synchronized boolean markAsUnused(Object item) {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (item == items[i]) {
        if (used[i]) {
          used[i] = false;
          return true;
        } else
          return false;
      }
    }
    return false;
  }

}
```

-clase BlockingQueue:

Una cola que además admite operaciones que esperan que la cola se vuelva no vacía al recuperar un elemento, y que espere a que haya espacio disponible en la cola cuando se almacena un elemento.
Los métodos de BlockingQueue vienen en cuatro formas, con diferentes formas de manejar operaciones que no pueden satisfacerse de inmediato, pero pueden satisfacerse en algún momento en el futuro: uno arroja una excepción, el segundo devuelve un valor especial (ya sea nulo o falso, dependiendo de operación), el tercero bloquea el subproceso actual indefinidamente hasta que la operación pueda tener éxito, y el cuarto bloquea solo un límite de tiempo máximo antes de darse por vencido.

```
public class Producer implements Runnable {

    private final BlockingQueue<Integer> queue;

    @Override
    public void run() {

        try {
            process();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

    }

    private void process() throws InterruptedException {

        // Put 20 ints into Queue
        for (int i = 0; i < 20; i++) {
            System.out.println("[Producer] Put : " + i);
```

```
            queue.put(i);
            System.out.println("[Producer] Queue remainingCapacity : " +
queue.remainingCapacity());
            Thread.sleep(100);
        }

    }

    public Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }
}
```

-Clase AtomicInteger:

Un valor int que puede actualizarse atómicamente. Consulte la especificación del paquete java.util.concurrent.atomic para obtener una descripción de las propiedades de las variables atómicas. Un AtomicInteger se usa en aplicaciones tales como contadores incrementados atómicamente, y no se puede usar como reemplazo de un Integer. Sin embargo, esta clase sí extiende el Número para permitir el acceso uniforme de herramientas y utilidades que se ocupan de clases basadas en números.

```
AtomicInteger atomicInteger = new AtomicInteger(123);

int theValue = atomicInteger.get();
```