Un proceso puede tener subprocesos o hilos, un hilo es una secuencia de código en ejecución dentro del contexto de un proceso que ejecuta sus instrucciones de forma independiente y necesita la supervisión de un proceso padre. Los hilos comparten el contexto del proceso, pero cada hilo tiene una parte local (su contador de programa, registros del CPU y pila para saber dónde se está ejecutando), dentro de un proceso podemos tener varios hilos de ejecución colaborando en la ejecución del proceso. Los hilos son la secuencia más pequeña de instrucciones que el Sistema Operativo puede manejar de forma independiente.

Los hilos comparten el espacio de memoria del usuario, los datos y los espacios de direcciones y los procesos poseen espacio de memoria independiente e interactúan a través de mecanismos de comunicación del sistema.

Usamos hilos para programas que realicen varias tareas de forma simultánea. Ej: En un procesador de texto podemos tener un hilo para guardar los datos cada x tiempo y otro para la corrección ortográfica automática.

La ventaja de la multitarea es:

a) Capacidad de respuesta, los hilos permiten a los procesos continuar atendiendo peticiones del usuario, aunque alguna de las tareas que se están realizando sea larga. Como sucede en un servidor Web puede tener un hilo para todas las peticiones de sus clientes y un hilo para responder a cada uno de los clientes que estén conectados o en un navegador podemos tener varias páginas cargándose de forma simultánea.
b) Compartición de recursos, los hilos comparten la memoria y todos los recursos del proceso al que pertenecen. Como pueden acceder y modificar los datos al mismo tiempo necesitan de medios de sincronización para evitar problemas de acceso.
c) Como los hilos utilizan la misma memoria del proceso, la creación de nuevos hilos no requiere memoria adicional, consume menos memoria que los procesos.
d) Paralelismo real, los hilos nos permiten aprovechar los núcleos del procesador ejecutándose varias instrucciones a la vez (una por núcleo).
e) La gestión de los hilos por el sistema es más rápida que la de procesos se tarda menos en conmutar hilos de un proceso que entre procesos.

ESTADOS DE UN HILO

Al igual que los procesos los hilos pueden cambiar de estado a lo largo de la ejecución. Los estados posibles son:

- **Nuevo**, el hilo está preparado para ejecutarse, pero no se ha realizado la llamada para la ejecución del código del programa.
- **Listo**, aun no le ha sido asignado un procesador o núcleo para ejecutarse.
- Se puede ejecutar (**Runnable**), el hilo está ya ejecutándose.
- **Bloqueado**, puede está esperando una operación de E/S, dormido, suspendido,… esperando volver a ejecutarse.
- **Terminado**, el hilo ha terminado su ejecución y no libera ningún recurso ya que este pertenece al proceso y no al hilo. El hilo puede terminar o el propio proceso lo finaliza.

OPERACIONES DE LOS HILOS

a) **Creación de un hilo**, cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución. Este hilo puede crear a su vez nuevos hilos que ejecutarán código diferente o tareas.

Podemos crear hilos o hebras (**Thread**) en Java por medio de la interface **Runnable**, o creando una clase **Thread**, ambas pertenecen a la librería **java.lang**.

Con la interfaz **Runnable**, con el método **run()** se crea el hilo y contiene el código a ejecutarse. La utilizamos si solo empleamos el método run.

Para cualquier otro caso utilizamos la clase **Thread**, donde podemos utilizar otros métodos. Java no soporta herencia múltiple de forma directa. El método que empleamos para crear el hilo es **start()**, que llama indirectamente al método run.

Para crear un hilo con Runnable se crea una clase que implemente la interfaz, solo con el método run con la tarea a realizar, además debemos crear una instancia de Thread dentro de la nueva clase. Ejemplo1

b) **Espera de un hilo**, la ejecución de un hilo se puede suspender con **join()** esperando hasta que el hilo correspondiente finalice su ejecución. O podemos dormir un hilo con **sleep(milisegundos)** por el tiempo indicado.

join( ) join(milisegundos) join(milisegundos,nanosegundos) sleep(milisegundos) sleep(milisegundos, nanosegundos)

Una interrupción es una indicación a un hilo que debería de dejar de hacer lo que está haciendo. Un hilo manda una interrupción mediante el método **interrupt()** y recibe la excepción InterruptedException. Cuando se produce una interrupción el programador debe decidir qué hacer, generalmente se termina la ejecución.

La excepción InterruptedException, se puede dar con **sleep** y con **join**.

Para ver todos los estados posibles de una hebra, podemos ejecutar:

```
for(Thread.State estado : Thread.State.values())
  System.out.println(estado);
```

En la ejecución de mi programa yo podre preguntar por el estado actual de la hebra para que suceda algún evento.

```
If (hilo.getState() != Thread.State.TERMINATED)
```

El método **getState()**, nos devuelve el estado de la hebra.

El método **isAlive()**, es para comprobar si la hebra no ha finalizado su ejecución antes de ponernos a trabajar con él. Devuelve un booleano, true si está vivo el hilo.

```
boolean isAlive( )
```

El método **toString()**, nos devuelve el nombre del hilo, la prioridad y el grupo al que pertenece. Con **currentThread()** obtendríamos la misma información.

Ejemplo: hilo.toString() Thread[Thread-0,5,main]

getId(), nos devuelve un entero que el número de identificador del hilo. No es el mismo número que nos devuelve al ejecutar una orden del sistema.

getName( ), nos daría solo el nombre del hilo. Thread-?.

getPriority(), nos indica la prioridad actual del hilo .

setName(String), podemos cambiarle el nombre al hilo

setPriority(int), podemos cambiarle la prioridad al hilo. Podemos utilizar las constantes **MAX_PRIORITY**, es la mayor prioridad que puede tomar cuyo valor es 10, **NORM_PRIORITY** es la prioridad normal y toma el valor 5 y **MIN_PRIORITY** es la menor prioridad y toma el valor 1. Las usamos como Thread.MAX_PRIORITY.
interrupted(), comprueba si el hilo ha sido interrumpido.
setDaemon(boolean), para lanzar una hebra de servicio (demonio). No debe ejecutarse después de haber sido lanzada la hebra.
isDaemon(), nos dice si una hebra es un demonio.

PROBAR
activeCount(), nos devuelve el número de hilos que tenemos incluye los bloqueados y los que están en espera.
Podemos declarar un array de hilos y lanzarlos:
Thread [] tabla = new Thread[entero];
Thread.enumerate(tabla); lanzaría los hilos de la tabla.

AQUÍ

PLANIFICACIÓN DE HILOS

Cuando se trabaja con hilos, es necesario planificar los hilos para asegurarse de que pueden ejecutarse. El planificador del Sistema Operativo es el que indica que proceso se va a ejecutar y el hilo a ejecutarse está en función de los núcleos que tengamos. Java por defecto utiliza un planificador apropiativo, se ejecutará el de máxima prioridad, a igualdad de prioridades el planificador elige y empleará tiempo compartido si el Sistema Operativo lo permite.
Para dar prioridad a los hilos tenemos el método **setPriority()**, los valores van de 1 al 10, siendo 10 el de máxima prioridad (Windows) y en Linux van del -20 a 19, el valor por defecto es 0 y el máximo es -20. Al crear un hilo se hereda la prioridad.

SICRONIZACIÓN DE HILOS

Los hilos se comunican con intercambio de información a través de variables y objetos en memoria. Los hilos de un proceso acceden a toda la memoria, las variables y objetos del mismo siendo así su comunicación más eficiente, pero si varios hilos manipulan de forma concurrente los objetos puede llevarnos a resultados erróneos o a paralizar el proceso.

PROBLEMAS DE SINCRONIZACIÓN

Condición de carrera, si el resultado de la ejecución de un programa depende del orden en que se realicen los accesos a memoria. Por ejemplo, si dos hilos modifican una variable a la vez una la incrementa y el otro la decrementa.

Inconsistencia de memoria, se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato.

Inanición, es cuando un proceso o hilo se le deniega el acceso a un recurso compartido, ya que todos los procesos existentes toman el recurso antes que él. Puede darse si todos los procesos que entren tienen prioridad superior a él.

Interbloqueo, se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado.

Bloqueo activo, es muy parecido al **interbloqueo,** pero ahora los procesos o hilos activos envueltos en el bloqueo cambian de estado con respecto al otro. Ejemplo de dos personas que se cruzan en una acera y ambos ceden el paso a la vez al otro o cambian de trayectoria para que el otro pase y se mueven en la misma dirección bloqueándose el paso.

MECANISMOS DE SINCRONIZACIÓN

La condición de carrera y la inconsistencia de memoria se producen porque se ejecutan varios hilos de forma concurrentemente pudiendo ser ordenados de forma diferente a la esperada. La solución consiste en que los hilos accedan a datos compartidos de forma ordenada o síncrona. Cuando ejecute código que no afecte a datos compartidos puede ejecutarse en paralelo o asíncrona. (DEKKER, DIJKSTRA,)

Algunos de los mecanismos de sincronización son:

a) **Condición de Bernstein**, dos segmentos i y j de código son independientes y pueden ejecutarse en paralelo de forma asíncrona en diferentes hilos, si cumplen:
   I. **Dependencia de flujo**, todas las variables de entrada del segmento j tienen que ser diferentes de las variables de salida del segmento i. Si no fuera así, el segmento j dependería de la ejecución de i.
   II. **Antidependencia**, todas las variables de entrada del segmento i tiene que ser diferentes de las variables de salida del segmento j. Es lo contrario de la condición anterior.
   III. **Dependencia de salida**, todas las variables del segmento i tienen que ser diferentes de las variables de salida del segmento j.

b) **Operación atómica**, es una operación que sucede toda al mismo tiempo. Se ejecutará de forma continuada y sin ser interrumpida, por ningún otro proceso o hilo, puede modificar datos relacionados mientras se esté realizando la operación. Ej: sacar dinero de un cajero si te da el dinero automáticamente lo resta de tu cuenta. Es más típico si tenemos un solo procesador (cores o núcleos). Si tenemos varios procesadores (cores o núcleos) deberíamos declarar las variables como volatile, al realizar los cambios de la variable en la memoria y no en los registros.

c) **Sección crítica**, es una región de código en la que se accede de forma ordenada a variables y recursos compartidos, de forma que se puede diferenciar de aquellas zonas de código que se pueden ejecutar de forma asíncrona. En este caso ningún otro proceso puede ejecutar su sección crítica. Si un proceso lo intentara se bloqueará hasta que termine el primer proceso de ejecutarse. Debemos diseñar un protocolo que permita a los procesos cooperar. Cualquier solución al problema de la sección crítica debe cumplir:

a. **Exclusión mutua**, si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica.
b. **Progreso**, si ningún proceso está ejecutando su sección crítica y hay varios procesos que quieren entrar en su sección crítica, solo aquellos procesos que están esperando para entrar pueden participar en la decisión de quien entra.
c. **Espera limitada**, debe existir un número limitado de veces que se permite a otros procesos entrar en su sección crítica, después de que otro proceso haya solicitado entrar en la suya y antes de que se le conceda. En caso contrario se producirá la inanición.

d) **Semáforos**, se pueden utilizar para controlar el acceso a un determinado recurso formado por un número finito de instancias. Se representa como una variable entera que representa el número de instancias disponibles del recurso compartido y una cola donde se almacenan los procesos o hilos bloqueados, esperando utilizar el recurso. En la fase de inicialización se da un valor al semáforo, igual al número de recursos inicialmente disponibles. Posteriormente podremos modificar ese valor mediante las operaciones:
   a. **wait (espera)**, si se ejecuta esta operación se disminuye el número de instancias disponibles en uno, ya que se supone que se va a utilizar. Si el valor es menor que cero no hay instancias disponibles, el proceso quedará en estado bloqueado hasta que el semáforo se libere. El valor negativo del semáforo indica cuántos procesos están bloqueados esperando el recurso.
   b. **signal (señal)**, un proceso cuando termina de usar la instancia del recurso compartido avisa de su liberación mediante esta operación. Ello aumenta el valor de instancias disponibles en el semáforo. Si el valor del semáforo es menor que cero despertara a un proceso bloqueado de la cola. El hilo a despertar puede ser aleatorio.

   Las operaciones **wait (acquire)** y **signal (release)** deben ser atómicas para evitar los problemas anteriores.

   En java la utilización de semáforos se realiza mediante el paquete java.util.concurrent y su clase Semaphore.

e) **Monitores**, es un conjunto de métodos atómicos que proporcionan de forma sencilla exclusión mutua a un recurso. Los métodos indicados permiten que cuando un hilo ejecute uno de los mismos, solamente ese hilo pueda estar ejecutando un método monitor. Es similar a los semáforos, pero el programador lo único que tiene que hacer es ejecutar una entrada de monitor. Un monitor no puede ser utilizado de forma incorrecta, y los semáforos dependerán del programador. En java utilizamos la palabra **synchronized**, sobre la región de código para indicar que se debe ejecutar como si de una sección crítica se tratara. Podemos utilizar esta palabra con los métodos y con las sentencias sincronizadas. Los métodos sincronizados son un mecanismo para crear una sección crítica de forma sencilla. La ejecución por parte de un hilo de un método sincronizado de un objeto en java, imposibilita que se ejecute a la vez otro método sincronizado del mismo objeto por parte de otro hilo, cumpliendo así con los requisitos de las secciones críticas. Cuando un hilo invoca a un método sincronizado, adquiere automáticamente el monitor que el sistema crea específicamente para todo el objeto

que contiene ese método, ningún otro hilo podrá ejecutar ningún método sincronizado del mismo objeto, mientras el monitor de ese objeto no sea liberado.
Para crear un método sincronizado solo es necesario añadir la palabra **synchronized** en la declaración del método, los constructores son síncronos por defecto y no pueden llevar esta palabra.

f)

……

Herramientas para la monitorización de Java:

- **JVisualVM** es una herramienta grafica que nos permite monitorizar la máquina virtual creada y de los distintos procesos que se ejecutan en Java. Las gráficas que obtenemos son en tiempo real. También podemos conectarnos a una máquina remota. La utilidad de esta herramienta es:
  - Conocer el uso de los recursos de memoria, CPU que consumen las aplicaciones.
  - Identificar que partes de las aplicaciones consumen más recursos.
  - Conocer los hilos que están en ejecución.

  http://visualvm.java.net/

  Se instala desde Linux desde el soporte de Ubuntu o desde una terminal.

  sudo apt-get install visualvm (es sin la jota delante)
  Si diera problemas editamos el fichero y ponemos una línea de comentario:
  sudo gedit /usr/lib/visualvm/etc/visualvm.conf
  # jdkhome="/usr/lib/jvm/java-6-openjdk-i386"
  Se ejecuta desde la línea de comando con jvisualvm.

- **Jconsole**, está instalado por defecto. Lo ejecutamos llamando a jconsole desde una terminal. Herramienta gráfica. ¿Solo para elementos de Java?
- **Jstack**, está instalada por defecto y se ejecuta desde la línea de comando pero debemos dar el número de pid. No es una herramienta gráfica muestra texto en la pantalla. ¿Solo para procesos de Java?
- **Netbeans Profiles** https://profiler.netbeans.org/download/ Ubuntu y Windows
  https://netbeans.org/download/magazine/01/nb01_profiler.pdf
- **JProfiler.** http://www.ej-technologies.com/download/jprofiler/files descarga
  http://resources.ej-technologies.com/jprofiler/help/doc/help.pdf
- **Yourkit**. http://www.yourkit.com/download/
- **JProbe**. http://jprobe.software.informer.com/8.3/ descarga
  http://us-downloads.quest.com/Repository/support.quest.com/JProbe/8.3/Documentation/JProbe_Tutorials.pdf

GPU(graphics processing unit) las tarjetas gráficas actuales tienen un procesador para agilizar el trabajo al procesador, pueden tener hasta 100 núcleos, pero sus operaciones están limitadas , todos realizan la tarea pero con distintos datos.

Para invocar métodos de forma remota, para aplicaciones cliente-servidor se utiliza Java RMI(Remote Method Invocation). Tenemos métodos: finalize, wait, notify, notifyall que vienen de la clase Object.

EJEMPLOS DE HILOS
http://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CCQQFjAA&url=http%3A%2F%2Fjaviergarbedo.es%2F10-apuntes-java%2Fhilos-de-ejecucion%2F38-hilos-de-ejecucion&ei=_5pjVJn0OYnSaJSsgOAN&usg=AFQjCNG9AlG4oF1fxLOzCFHFDKsxPpGI-w&bvm=bv.79189006,d.d2s