# 

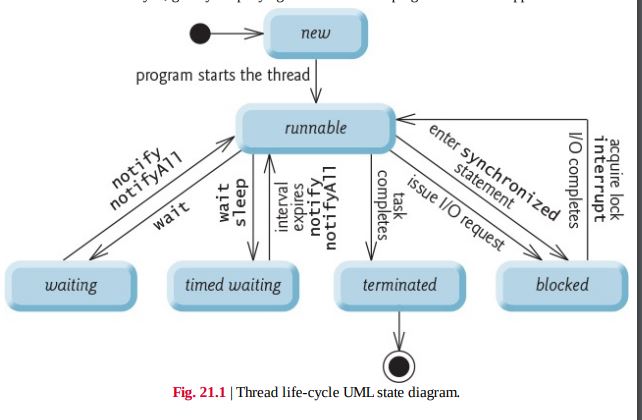
**Proceso**: instancia de un programa en ejecución

Cada proceso tiene los recursos necesarios para ejecutar un programa. Un proceso tiene un espacio de direcciones virtual, manejadores a objetos del sistema, un contexto de seguridad, un identificador de proceso único (PID), variables de entorno, una clase de prioridad, un espacio de memoria de trabajo mínimo y máximo, y al menos un hilo de ejecución.

* Cada proceso es iniciado con un único hilo, llamado típicamente el hilo principal, pero puede crear hilos adicionales a partir de cualquiera de sus hilos.
* Algunos SSOO usan el término "tarea (task)" para referirse a un programa en ejecución
* Un proceso siempre se almacena en la memoria principal (RAM)
* Varios procesos pueden estar asociados al mismo programa. Ejemplo: varias instancias de ejecución de la calculadora de Windows.
* En un sistema multiprocesador, varios procesos pueden ser ejecutados en paralelo
* En un sistema uniprocesador, no es posible conseguir paralelismo real pero se puede usar un algoritmo de planificación de procesos para ejecutar varios procesos de forma concurrente obteniendo para el usuario la ilusión de un entorno de ejecución de paralelismo real

**Hilo**: entidad perteneciente a un proceso que puede ser planificada para ser ejecutada

* Todos los hilos de un proceso comparten el mismo espacio de direcciones virtuales y recursos del sistema. Además, cada hilo tiene manejadores de excepciones, una prioridad de planificación, un espacio de almacenamiento local del hilo, un identificador único del hilo (Thread ID), y un conjunto de estructuras que usará el sistema para almacenar el contexto del hilo hasta que sea planificado.
* El contexto del hilo incluye el conjunto del registros a nivel máquina del hilo, la pila del núcleo (kernel stack), un bloque de entorno del hilo, y una pila del usuario en el espacio de direcciones del proceso al que pertenece el hilo.
* Los hilos pueden tener su propio contexto de seguridad
* Un hilo es un subconjunto de un proceso
* Un hilo es un "proceso ligero", ya que es similar a un proceso real pero ejecutado en el contexto del proceso al que pertenece y compartiendo los mismos recursos de los que dispone el proceso (que son asignados por el núcleo al proceso)
* Múltiples hilos pueden aprovechar el paralelismo real de sistemas multiprocesador
* En sistemas uniprocesador, se usa un algoritmo de planificación de hilos para ejecutar hilos en el procesador (hilo a hilo, sólo un hilo a la vez)
* Debido a que los hilos de un proceso comparten la misma memoria, es de vital importancia la sincronización del acceso a los datos compartidos dentro del proceso



**Tarea**: conjunto de instrucciones de un programa cargadas en memoria que puede ser planificado como una unidad de trabajo.

* El término tarea se usa principalmente en el contexto de planificación para referirse a un conjunto de instrucciones de un hilo o de un proceso que puede ser planificado para ser ejecutado en un procesador.
* Desde el punto de vista del procesador hay poca diferencia entre tarea de un hilo o de un proceso.
* En el contexto de Java, las tareas a planificar son el conjunto de sentencias dentro de un bloque de un método run() de una clase que sea subtipo de Runnable. Dichas tareas serán asignadas para ejecutarse en hilos los cuales pueden pertenecer típicamente a un grupo de hilos de trabajo (*pool of worker threads*). Ver Executor
* Cualquier método llamado desde el método run() de la clase que implementa Runnable, se ejecutará como parte de la tarea del hilo en ejecución.
* **Paralelismo**: simultáneamente, al mismo tiempo(no se pueden ejecutar 2 programas en paralelo con un procesador que sea mononúcleo y monohilo, que solo pueda seguir una instrucción por ciclo)
* **Concurrente**: pasa de un proceso a otro rápidamente , simulando el paralelismo
* [Clase Process](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Process.html)
* [Clase Thread](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Thread.html)
* [Interface ExecutorService](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html): un objeto "ejecutor", subtipo de ExecutorService puede ser "apagado" (shutdown), lo cual hará que rechace nuevas tareas. El método *shutdown()* permitirá que las tareas que hayan sido recibidas previamente sean ejecutadas antes del "apagado", mientras que el método \*shutdownNow() evita que las tareas que estén esperando para ejecutarse se empiecen a ejecutar e intenta parar las tareas que se estén ejecutando actualmente. Tras la finalización, un objeto "ejecutor" no tiene ninguna tarea ejecutándose activamente, ni tampoco ninguna esperando a ser ejecutada y no es posible enviar más tareas al "ejecutor". Un objeto "ejecutor" (subtipo de ExecutorService) debe ser "apagado" para permitir la reclamación de sus recursos.

*Seguridad para hilos (thread-safe)*

Una clase es segura para hilos si se comporta correctamente cuando es accedida (ella y los objetos que se instancian a partir de ella) desde múltiples hilos, con independencia de la planificación o el entrelazamiento de la ejecución de esos hilos por el entorno de ejecución, y sin usar ninguna sincronización adicional u otra coordinación en la parte del código llamante.

*Sincronización de hilos*

* La **sincronización de hilos** es el procedimiento por el cual se realiza la coordinación del acceso a datos compartidos por múltiples hilos.
* Mediante la sincronización de hilos podemos asegurar que cada hilo que accede a un objeto compartido excluye al resto de hilos de hacer eso mismo de forma simultánea. A esto se le llama **exclusión mutua**
* En realidad, la sincronización de datos es necesaria únicamente para datos compartidos mutables (modificables), es decir, datos que pueden cambiar a lo largo de su ciclo de vida. Con datos inmutables (no modificables) que por tanto no cambian, no provocan problemas
* La palabra clave **synchronized** puede ser usada a distintos niveles:  
  -- Métodos de instancia  
  -- Métodos estáticos  
  -- Bloques de código

Cuando usamos un bloque sincronizado, internamente Java usa un monitor llamado monitor de cerrojo (*monitor lock*) o cerrojo intrínseco (*intrinsic lock*) para proporcionar sincronización. El **monitor** está asociado a un objeto, por tanto, todos los bloques sincronizados del mismo objeto sólo pueden tener un único hilo ejecutándolos al mismo tiempo. El monitor asegura que cómo máximo 1 hilo *está en posesión* del cerrojo del monitor en todo momento.

Por tanto, la **exclusión mutua** se puede conseguir mediante los monitores y los cerrojos de monitores.

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

En el caso de *sincronización de bloques de código sobre un objeto*, sólo un hilo podrá ejecutar el código de la sección crítica sobre el objeto. Si hay varios hilos que intentan a la vez ejecutar dicho bloque crítico a la vez, sólo un hilo estará activo sobre el objeto y el resto de hilos que intenta ejecutar dicho bloque de código sobre el objeto quedarán en estado bloqueado.

*object* es el objeto cuyo cerrojo debe ser adquirido para ejecutar el bloque de código de la sección crítica. *object* podría ser *this*

synchronized (object) {  
// Sección crítica  
}

Cuando el hilo que posee el cerrojo sobre el objeto finaliza la ejecución de la sección crítica, libera el cerrojo y uno de los hilos que están intentando entrar (ejecutar) dicho bloque crítico puede adquirir el cerrojo y proceder a ejecutar dicho código crítico.

Java también permite *métodos sincronizados*.  
Antes de ejecutarse, un método de instancia sincronizado, debe adquirir el cerrojo del objeto al cual pertenece.

*De forma similar, un método estático sincronizado debe adquirir el cerrojo sobre el objeto de tipo Class que representa la clase en la cual el método está declarado. Un objeto de tipo Class es la representación en tiempo de ejecución de una clase que ha sido cargada por la máquina virtual de Java (JVM) en memoria.*

public synchronized void metodo()  
{  
// Sección crítica  
}

*Sleep*

Sleep:

* Sirve para pausar la ejecución del hilo actual normalmente por un tiempo o hasta recibir una interrupción. El hilo pasa a estado de espera. Vuelve a ejecución cuando expira el tiempo especificado o bien recibe una interrupción.
* El tiempo real durante el cual un proceso duerme depende de los temporizadores del sistema y del planificador. Para un sistema “tranquilo” el tiempo real será cercano al tiempo especificado al invocar a sleep(...) pero en un sistema “cargado” será algo más.
* Ojo: si usamos sleep() sin tiempo **dentro de la sección crítica**, no se liberaría el cerrojo con lo cual ningún otro hilo podría entrar en la sección crítica y el programa no podría continuar.
* Si usamos sleep(tiempo) **dentro de la sección crítica** el hilo actual seguiría en posesión del cerrojo y dormiría ese tiempo. Al volver a ejecución, seguiría ejecutándose y sólo al finalizar de ejecutarse liberaría el cerrojo y otro hilo podría ejecutar ese método sincronizado sobre el mismo objeto. Ver método sincronizado add(...) de la clase