Tema 2. Procesos vs. hilos

1. Procesos y Ciclo Vida

- El ciclo de vida de un proceso es el siguiente:

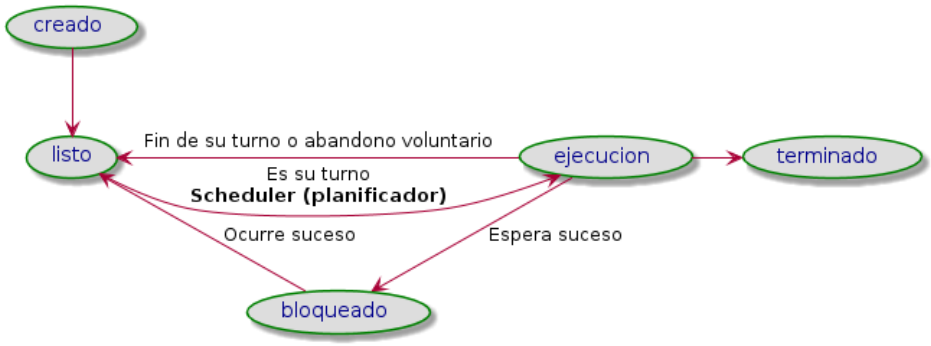

Figure 1: Ciclo de vida.

- El encargado de darle la oportunidad de usar la CPU es el *Planificador de procesos* o Scheduler → forma parte del núcleo del SO. En este otro enlace tienes más información sobre el planifcador del núcleo Linux.
- Una forma bastante justa y extendida es hacerlo mediante asignación de rodajas de tiempo:
  - Cuando un proceso cumple su tiempo de permanencia en el procesador, éste es desalojado y pasado a *Listo*. Esperará una nueva oportunidad para pasar a ejecución. También puede abandonar voluntariamente la CPU

- El acto de cambiar un proceso de estado se llama *Cambio de contexto*. Se trata de una operación *costosa*

- En un SO tradicional, la memoria se divide en:
  Espacio de usuario :

en él se encuentra la mayor parte de la información relativa a los procesos de usuario

Espacio de núcleo :
en él reside el código y las estructuras propias del sistema operativo.

- La información relativa a un proceso suele estar dividida entre los dos espacios.
- La parte del espacio del núcleo contiene lo que se conoce como **bloque de control del proceso**

2. Procesos en Unix con C

- En Unix todos los procesos, excepto el primero (el número `0`), se crean con una llamada a `fork()`. Puedes ver un contraargumento al uso de `fork` en este artículo. Puedes ver toda la información relativa a `fork` con la orden: `man 2 fork` o también aquí.
- Para optimizar la creación del nuevo proceso se emplean técnicas como Copy On Write (COW).
- El proceso que invoca a `fork` es el *proceso padre*.
- El proceso creado es el *proceso hijo*.
- El proceso `0` se crea en el arranque. Hace una llamada a `fork` para crear el proceso `1`, `init`, y a continuación se convierte en el proceso "*intercambiador de procesos*".
- Podemos observar la tabla de procesos activos con instrucciones como `top`, `htop`, `ps`, etc...

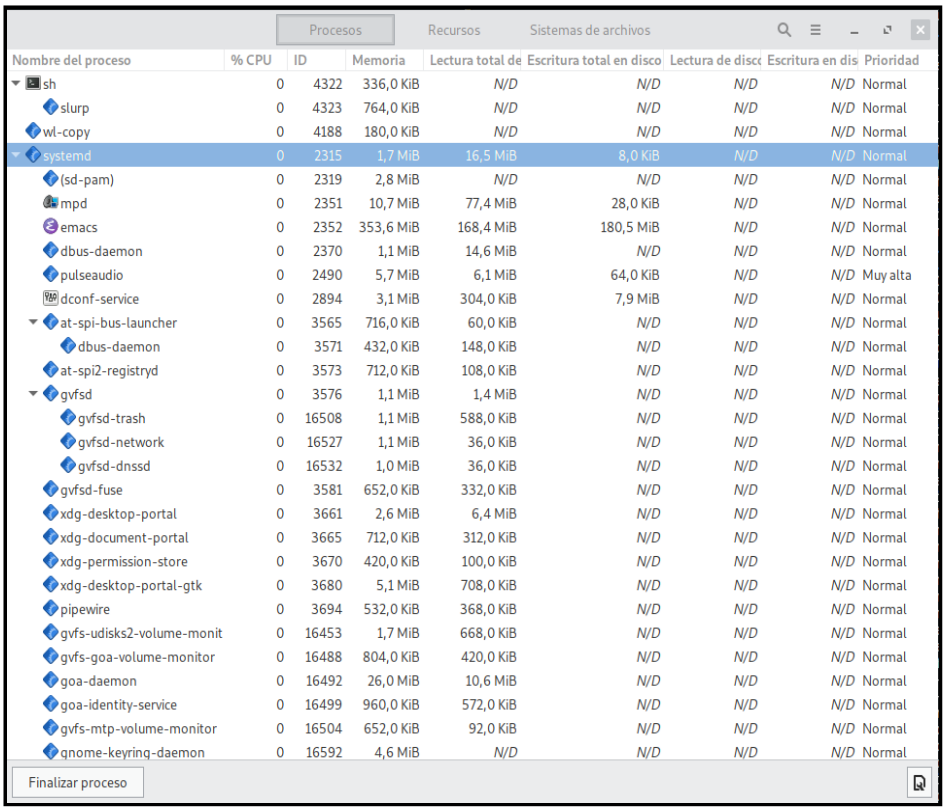

Figure 2: Ciclo de vida.

```
1: int pid;
2:
3: if ( (pid = fork()) == -1 )
4:   perror ("Error en la llamada a fork");
5: else if (pid == 0)
6:   // código que ejecutará el proceso hijo
7: else
8:   // código que ejecutará el proceso padre
```

- La llamada a fork duplica todo el contexto del proceso. Es interesante que conozcas el uso *idiomático* de `fork-exec`. Para todo ello, echa un vistazo a este vídeo.
- Todas las variables, incluidas las globales y las estáticas, son inaccesibles para el otro proceso: compartir información es complicado (mecanismos IPC)

2.1. Un ejemplo

```
1: /************/
2: /* procesos */
3: /************/
4: #include <sys/types.h>
5: #include <unistd.h>
6: #include <sys/wait.h>
7: #include <stdlib.h>
8: #include <stdio.h>
9:
10: #define NUM_PROCESOS 5
11: int I = 0;
12:
13: void codigo_del_proceso (int id) {
14:   int i;
15:   for (i = 0; i < 50; i++)
16:     printf("Proceso %d: i = %d, I = %d\n", id, i, I++);
17:   exit(id); // el id se almacena en los bits 8 al 15 antes de
18:             // devolverlo al padre
19: }
20:
21: int main() {
22:   int p;
23:   int id [NUM_PROCESOS] = {1,2,3,4,5};
24:   int pid;
25:   int salida;
26:
27:   for (p = 0; p < NUM_PROCESOS; p++) {
28:     pid = fork ();
29:     if (pid == -1) {
30:       perror ("Error al crear un proceso: ");
31:       exit (-1);
32:     }
33:     else if (pid == 0) // Codigo del hijo
34:       codigo_del_proceso (id[p]);
35:     }
36:     // Codigo del padre
37:     for (p = 0; p < NUM_PROCESOS; p++) {
38:       pid = wait (&salida);
39:       printf("Proceso %d con id = %x (%x) terminado\n",
40:              pid, salida >> 8, WEXITSTATUS(salida));
41:     }
42: }
```

3. Hilos y Ciclo Vida

- Los hilos permiten concurrencia dentro de cada proceso

- Los procesos son entidades pesadas
  - la estructura del proceso está en la parte del núcleo, y cada vez que un proceso quiere acceder a ella tiene que hacer una llamada al sistema y consumir tiempo de procesador

- Los hilos son entidades ligeras: la estructura de hilos reside en el espacio de usuario.
  - Los hilos comparten la información del proceso, por lo que si un hilo modifica una variable de proceso, el resto de hilos verán esa modificación cuando accedan a esa variable.
  - El cambio de contexto entre hilos consume poco tiempo de procesador, de ahí su éxito.

4. Hilos y hardware

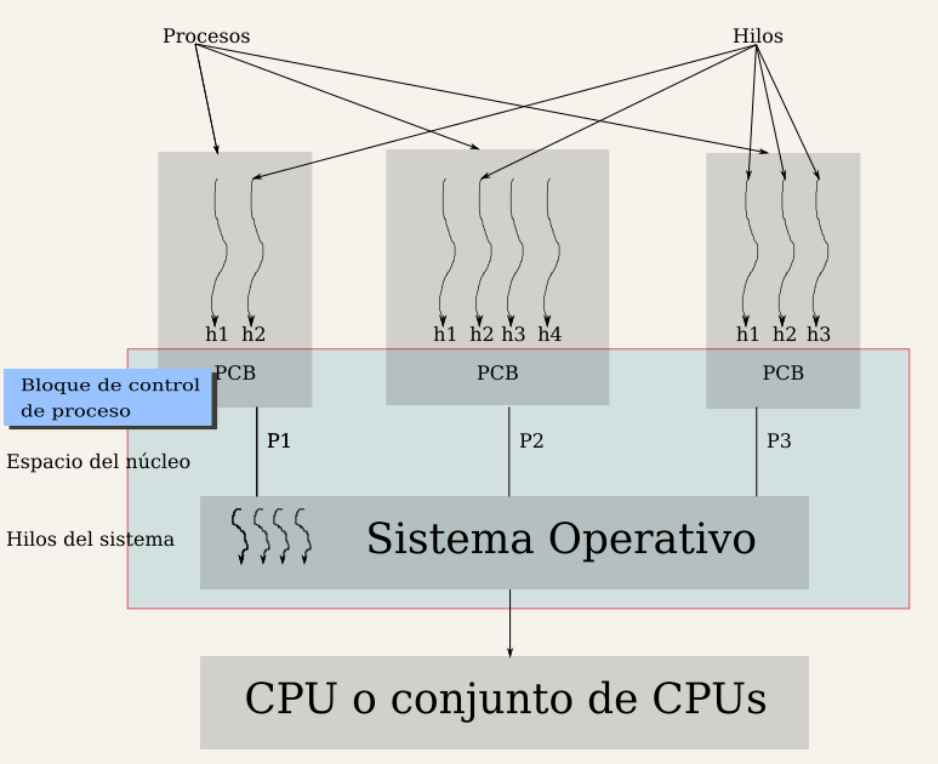

Figure 3: Hilos y hardware.

5. Dos niveles de hilos

- Hoy en día es normal que nos encontremos con dos 'niveles' de *hilos*
  - El que nos proporcione el lenguaje de programación empleado, p.e. Java
  - El que proporciona el SO.
  - Lo habitual es que el primero se *reescriba* mediante llamadas a este segundo -uno a uno, muchos a uno, muchos a muchos-.

6. Hilos y procesos ligeros

- Hoy en día los SO ofrecen el concepto de *proceso ligero* (LWP: *Light Weight Process*).
- Un *LWP* se ejecuta en espacio de usuario y esta sustentado por un *thread* o *hilo*.
- Un *LWP* comparte su espacio de direcciones y recursos del sistema con otros *LWP* que pueda crear el mismo proceso.
- Asociado al concepto de `hilo` aparece el de *almacenamiento local al hilo* -Thread Local Storage- o `TLS`. Lenguajes como D lo usan por defecto.

7. Hilos en Unix con C

- Se emplea la biblioteca `pthread` o POSIX threads.
- Es el interfaz más utilizado para implementar bibliotecas de hilos en entornos Unix.

```
#include <pthread.h>

int       pthread_create(...); // crear hilo
pthread_t pthread_self(void);  // Devuelve el ID dle hilo actual
void      pthread_exit(...);   // terminar hilo
int       pthread_join(...);   // espera por otro hilo
int       pthread_equal(...);  // comprueba si dos hilos son el mismo
```

- Los compiladores actuales de C/C++ como los de los proyectos GCC y LLVM incluyen lo que llaman *desinfectantes* de distintos tipos de errores cometidos al programar. Echa un vistazo aquí para GCC y

aquí para LLVM, en ambos casos busca las opciones que tienen que ver con *sanitize*. No solo hay para detectar errores cometidos al programar con *hilos* sino también para otro tipo de errores habituales.

8. Ejemplo con hilos POSIX

```c
/*********************************************/
/* hilos                                     */
/* compilación: cc -o hilos hilos.c -lpthread */
/*********************************************/

#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define NUM_HILOS 5
int I = 0;

void *codigo_del_hilo (void *id) {
  int i;
  for( i = 0; i < 50; i++)
    printf("Hilo %d: i = %d, I = %d\n", *(int *)id, i, I++);
  pthread_exit (id);
}

int main() {
  int h;
  pthread_t hilos[NUM_HILOS];
  int id[NUM_HILOS] = {1,2,3,4,5};
  int error;
  int *salida;

  for(h = 0; h < NUM_HILOS; h++) {
    error = pthread_create( &hilos[h], NULL, codigo_del_hilo, &id[h]);

    if (error){
      fprintf (stderr, "Error: %d: %s\n", error, strerror (error));
      exit(-1);
    }
  }

  for(h =0; h < NUM_HILOS; h++) {
    error = pthread_join(hilos[h], (void **)&salida);
    if (error)
      fprintf (stderr, "Error: %d: %s\n", error, strerror (error));
    else
      printf ("Hilo %d terminado\n", *salida);
  }
}
```

9. Ejemplo con hilos Python

- Usaremos pseudocódigo estilo Python, pero veamos cómo es un programa concurrente completo en este lenguaje

```
#! /usr/bin/env python

import threading

THREADS = 2
MAX_COUNT = 10000000

counter = 0

def thread():
        global counter

        print("Thread {}".format(threading.current_thread().name))

        for i in range(MAX_COUNT//THREADS):
                counter += 1

def main():
        threads = []

        for i in range(THREADS):
                # Create new threads
                t = threading.Thread(target=thread)
                threads.append(t)
                t.start() # start the thread

        # Wait for all threads to complete
        for t in threads:
                t.join()

        print("Counter value: {} Expected: {}\n".format(counter, MAX_CO

if __name__ == "__main__":
        main()
```

10. Ejemplo con el API de Windows

- La función principal de creación de hilos en el `API` de *Windows* es CreateThread.

```
#include <windows.h>
#include <iostream>

using namespace std;

DWORD Cont=0; // Variable compartida

DWORD WINAPI incrementar(LPVOID param)
{
  DWORD n = *(DWORD*)param;
  //int i;

  for( unsigned i = 0 ; i < n  ; i++)
  {
    Cont++;
    cout << "Contador sumando = " << Cont << "\n";
  }
  return 0;
}

DWORD WINAPI decrementar(LPVOID param)
{
  DWORD n = *(DWORD*)param;

  for( unsigned i = 0 ; i < n  ; i++)
  {
    Cont--;
    cout << "Contador restando = " << Cont << "\n";
  }
  return 0;
}

int main(int argc, char *argv[]) {
    DWORD TIdi,TIdd;
    HANDLE THandlei,THandled;
    int param = 100;

    //Creamos dos threads
    THandlei = CreateThread(NULL,0,incrementar,&param,0,&TIdi);
    THandled = CreateThread(NULL,0,decrementar,&param,0,&TIdd);

    cout << "Contador = " << Cont << "\n";

    //Esperamos a que acaben todos los threads
    WaitForSingleObject(THandlei,INFINITE);
    WaitForSingleObject(THandled,INFINITE);

    //Eliminamos los threads
    CloseHandle(THandlei);
    CloseHandle(THandled);

```

```
52:     cout << "Contador = " << Cont << "\n";
53:
54:     system("PAUSE");
55: }
```

11. Concurrencia en Java

11.1. Hilos en Java: ciclo de vida

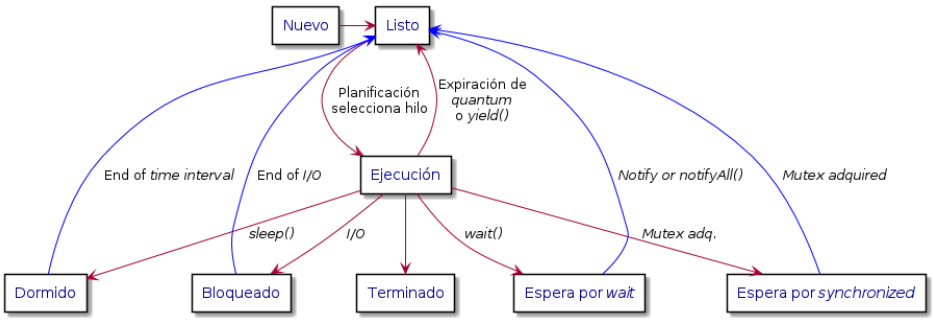

Figure 4: Ciclo de vida en Java.

11.2. Hilos y objetos

- Los hilos se representan en Java mediante la clase *Thread*.
- Sus métodos junto con algunos de la clase Object nos permiten un manejo completo de los hilos.
- Para cada programa Java existe un hilo de ejecución denominado hilo principal.
- Diferencia entre objeto e hilo:
  - Un objeto es algo estático, con una serie de atributos y métodos.
  - Pero quien ejecuta esos métodos es el hilo de ejecución.

11.3. Creación de hilos

- Clase Thread de Java
- Dos posibilidades:
  - Heredar de *Thread*

    - Implementar la interfaz *Runnable*

- En ambos casos hay que definir el método `run()`.

11.4. Creación de hilos: método run

- Contiene el código del hilo
- Método invocado cuando se ejecuta el hilo
- El hilo termina cuando termina `run`

11.5. Creación de hilos: ejemplos

- Heredando de Thread y redefiniendo el método `run`

```
public class Filosofo extends Thread {
        ...
        public void run() { ... }
        ...
}
```

- Implementando la interfaz `Runnable`

```
public class Filosofo implements Runnable {
        ...
        public void run() { ... }
        ...
}
```

- Creación y ejecución

```
//creación
Thread filosofo = new Filosofo();             //herencia
Thread filosofo = new Thread(new Filosofo()); //interface

//ejecución para ambos casos
filosofo.start();
```

11.6. Heredando de Thread

- Heredando de Thread y redefiniendo el método `run`

```
1: class ThreadConHerencia extends Thread {
2:      String palabra;
3:
4:      public ThreadConHerencia (String p) {
5:              palabra=p;
6:      }
7:
8:      public void run() {
9:              for (int i=0; i < 10 ; i++) {
10:                     System.out.print (palabra);
11:             }
12:     }
13:
14:     public static void main(String[]args){
15:             Thread a=new ThreadConHerencia(“hilo1”);
16:             Thread b=new ThreadConHerencia(“hilo2”);
17:
18:             a.start();
19:             b.start();
20:             System.out.println(“Fin del hilo principal”);
21:     }
22: }
```

- Se intercalan las salidas de los tres hilos creados, recordemos que tenemos el hilo principal y los dos creados

11.7. Implementando la interfaz Runnable

- Implementamos la interfaz `Runnable`, esta interfaz sólo tiene un método con la signatura `public void run()`.
- Este método es el que como mínimo tenemos que implementar en la clase.

```
1: public class ThreadConRunnable implements Runnable {
2:   String palabra;
3:
4:   public ThreadConRunnable (String p){
5:     palabra=p;
6:   }
7:
8:   public void run() {
9:     for(int i=0;i<10;i++)
10:    System.out.print(palabra);
11:  }
```

- Hasta aquí simplemente hemos creado una clase. Al contrario que antes, los objetos de esta clase no serán hilos ya que no hemos heredado de Thread.

- Si queremos que el objeto de esta clase se ejecute como un hilo independiente debemos crear un objeto de la clase Thread y pasarle como parámetro el objeto donde queremos que empiece su ejecución ese hilo.

```
1: public static void main(String[]args){
2:      ThreadConRunnable a=new ThreadConRunnable(“hilo1”);
3:      ThreadConRunnable b=new ThreadConRunnable(“hilo2”);
4:
5:      Thread t1=new Thread (a);
6:      Thread t2=new Thread (b);
7:
8:      t1.start();
9:      t2.start();
10:
11:     System.out.println(“Fin del hilo principal”);
12: }
```

- Se invoca al método `start` de la clase *Thread* que será el que se encarga de invocar al método *run()* de los objetos **a** y **b** respectivamente

- Si comparamos ambos métodos, la segunda forma puede parecer más confusa.
- Sin embargo es más apropiada debido a que en Java no hay herencia múltiple, al utilizar la primera opción nuestra clase ya no podría

heredar de otras clases.
- Si necesitamos que haya herencia de otras clases deberemos usar siempre la segunda opción.

11.8. Objeto autónomo en un hilo

- A veces necesitamos que un objeto autónomo se ejecute automáticamente en un nuevo hilo, sin intervención del cliente:

```
1: public class ObtejoAutonomo implements Runnable {
2:   private Thread hilo;
3:
4:   public ObjetoAutónomo() {
5:     hilo = new Thread(this);
6:     hilo.start();
7:   }
8:
9:   public void run() {
10:    if (hilo == Thread.currentThread()){
11:      //Hacer algo
12:    }
13:  }
14:
15:  //ATENCIÓN
16:
17:  public static void main(String []args){
18:    ObjetoAutónomo objeto = new ObjetoAutónomo();
19:  }
20: }
```

- Como vemos en este ejemplo, en la implementación del método run() hay que controlar cuál es el hilo que se está ejecutando y para ello nos servimos del método `currentThread()` de la clase `Thread`.
- Este método nos devuelve una referencia al hilo que está ejecutando ese código. Esto se hace para evitar que cualquier método de un hilo distinto haga una llamada a `run()` directamente.

11.9. Estado y propiedades de los hilos

- Método isAlive() para saber si un hilo está vivo o muerto
- Sistema de prioridades para el *scheduler* de la JVM:
  `setPriority(prioridad)`

- Método `yield()` para forzar la salida de un hilo de la CPU
- Otros métodos de utilidad: `wait()`, `notify`, `sleep(milisegundos)`

11.10. Planificación y prioridades

- Las prioridades de cada hilo en Java van de 1 (`MIN_PRIORITY`) a 10 (`MAX_PRIORITY`).
- La prioridad de un hilo inicialmente es la misma que la del hilo que lo creó.
- Por defecto, todo hilo tiene prioridad 5 (`NORM_PRIORITY`)
- La especificación de la máquina virtual no fuerza al uso de ningun algoritmo particular en la planificación de hebras.
- El planificador debe dar ventaja a las hebras con mayor prioridad.
- Si hay varias hebras con igual prioridad todas se deben ejecutar en algún momento.
- No se garantiza que hebras de prioridad baja pasen a ejecutarse si existe alguna hebra de mayor prioridad..., pero podría ser así.

- El código siguiente permite comprobar la implementación particular de nuestra máquina virtual

```
1: public class ComprobarPrioridad implements Runnable {
2:   int num;
3:   ComprobarPrioridad(int c) { num = c; }
4:
5:   public void run() {
6:     while (true)
7:     System.out.println(num);
8:   }
9:
10:  public static void main(String[] args) {
11:    Thread nueva;
12:    for (int c = 0; c < 10; c++) {
13:      nueva = new Thread(new ComprobarPrioridad(c));
14:      if (c == 0)
15:        nueva.setPriority(Thread.MAX_PRIORITY);
16:      nueva.start();
17:    }
18:  }
19: } // class
```

11.11. La clase Thread

- Atributos:
  - `public static final int MIN_PRIORITY`
  - `public static final int NORM_PRIORITY`
  - `public static final int MAX_PRIORITY`

- Constructores:
  public Thread():
  por defecto

  public Thread(String name):
  un nuevo hilo con nombre name

  public Thread(Runnable target):
  crea un nuevo hilo siendo target el que contiene el método run() que será invocado al lanzar el hilo con start()

  public Thread(Runnable target, String name):
  como el anterior, pero con nombre

- Métodos:
  public static Thread currentThread():
  retorna la referencia al hilo que se está ejecutando actualmente

  public String getName():
  retorna el nombre del hilo.

  int getPriority():
  retorna la prioridad del hilo

  public final boolean isAlive():
  chequea si el hilo está vivo

  public void run():
  contiene lo que el hilo debe hacer

- Métodos:
  public final void setName(String name):
  cambia el nombre del hilo por name

  public final void setPriority(int nuevaPrioridad):
  cambia la prioridad

  public static void sleep(long milis):

cesa la ejecución milis milisengudos

public void start():
hace que el hilo comience la ejecución

public static void yield ():
hace que el hilo que se está ejecutando actualmente pase a estado de listo, permitiendo a otro hilo ganar el procesador

public final void join():
Espera a que el hilo termine.

11.12. Ten en cuenta que en Java:

1. Las asignaciones entre tipos primitivos son atómicas. En el caso de `long` y `double` puede haber excepciones.
2. Las asignaciones de referencias son atómicas.
3. Las asignaciones de variables `volatile` son atómicas.
4. Todas las operaciones de las clases de `java.concurrent.Atomic*` son atómicas.

Es conveniento que consultes la especificación de la versión de Java que uses.

11.12.1. Aclaraciones

- **En ningún caso estas transparencias son la bibliografía de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.