Concurrencia en Java

¿Qué es un procesador?

Interpreta la conexión entre hardware y software.
Se encarga de interpretar de manera binaria, procesos matemáticos por medio de los cuales sean funcionales para el sistema.
Un procesador está compuesto por

Como distribuye el trabajo:

Multitasking/Multitarea

Maneja un cliente a la vez, donde si los clientes se van a conectar, establen una conexión unidireccional(No hay una respuesta de por parte de donde se realiza la solicitud),

Servidor iterativo

Tener una carga pequeña o media de solicitudes. Si hay muchas solicitudes, hay tiempo de espera más largo y hay que redistribuir procesos.

¿Qué son hilos?

Hilo

Asignar procesos a una clon del programa, que luego desaparece cuando se conecta con el principal.

En el host se realizan procesos independientes, atendiendo al cliente al mismo tiempo.

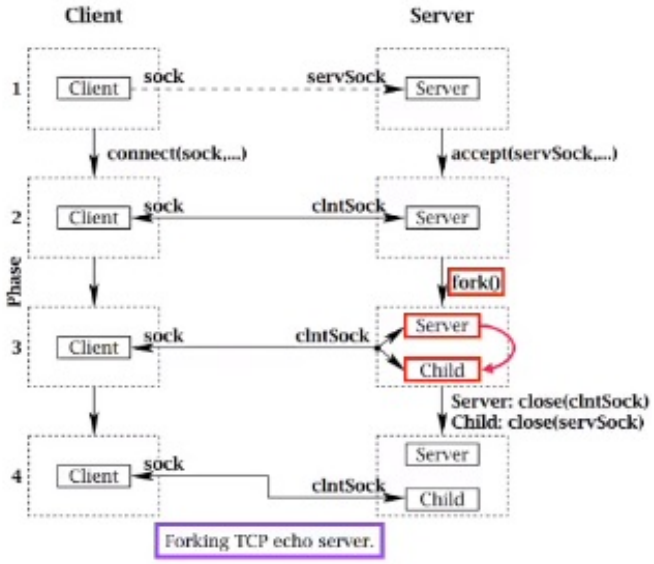
Es el proceso de ejecución dentro de un programa, un proceso es un conjunto de hilos(hacen un programa completo).

Socket

Una entrada para iniciar un procesos.

En UNIX(Linux) se tiene la función fork(), teniendo una clonación del proceso, se manejan de manera diferente.

Si el padre está ocupado, se crea un hijo(clon del padre) para distribuir la carga entre ambos y una vez completada, el hijo desaparece. Se distribuye para ayudar a cada cliente.



Sistema concurrente

Atender a cada cliente por medio de hilos

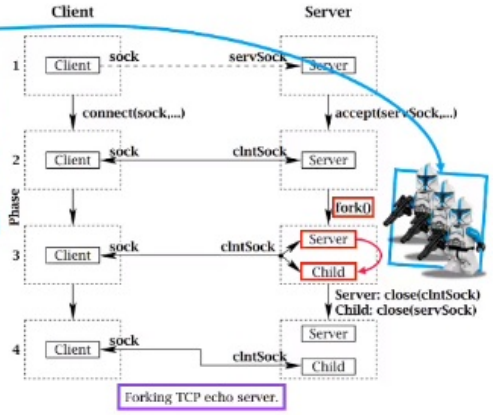
1. Llega solicitud
2. Padre crea al hijo (fork() Clonación)

3. El hijo da una respuesta de manera independiente sin afectar los otros procesos, tiene un coste computacional, porque es exactamente el mismo proceso(Hay que tener buenos servidores)
4. Padre mata al hijo una vez completado porque consume mucho(mata el proceso).

Proceso por-cliente

Ramificar (*fork*) un nuevo proceso para manejar cada cliente funciona, pero es costoso.

Cada vez que se crea un proceso, el sistema operativo debe duplicar todo el estado del proceso principal, incluida la memoria, la pila, los descriptores de archivo/socket, etc.

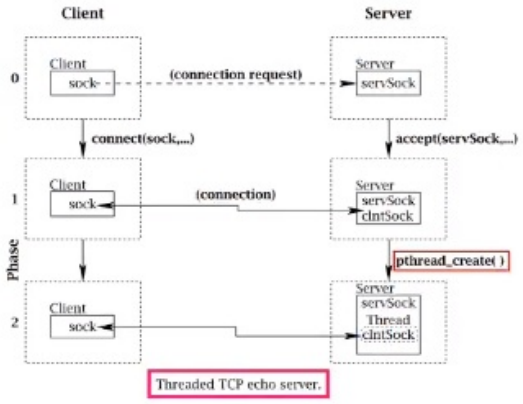


Cuando finaliza un proceso secundario, no desaparece automáticamente. En lenguaje UNIX, el hijo se convierte en *zombi*. Los *zombis* consumen recursos del sistema hasta que su padre los "cosecha" con una llamada a *waitpid()*.

Hilo por-cliente

Los hilos – *threads* – disminuyen este costo al permitir la multitarea dentro del mismo proceso: un hilo recién creado simplemente comparte el mismo espacio de direcciones (código y datos) con el padre, removiendo la necesidad de duplicar el estado principal.

El siguiente diagrama de ejemplo de un servidor simple *TCP Echo Server*, demuestra un enfoque multitarea hilo por cliente.



Procesos e hilos
Processes and threads

En la década de 1980, se descubrió que la noción tradicional del sistema operativo de un proceso que ejecuta una sola actividad no estaba a la altura de los requisitos de los sistemas distribuidos, y también de las aplicaciones más sofisticadas de una sola computadora que requieren concurrencia interna.

La solución a la que se llegó fue potenciar la noción de proceso para que pudiera asociarse a múltiples actividades. Hoy en día, un proceso consta de un entorno de ejecución junto con uno o más hilos – *threads* –.

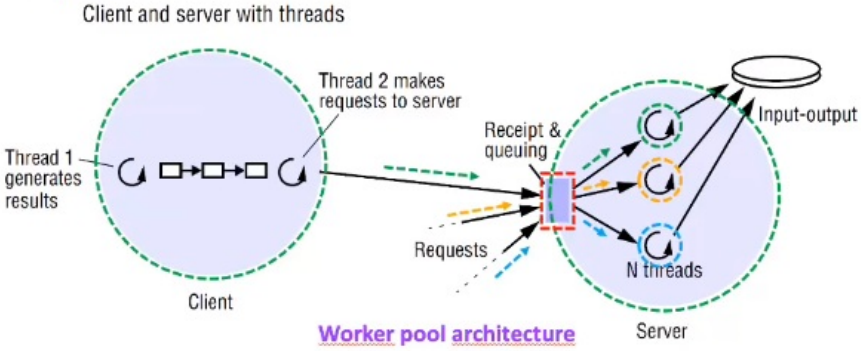
Un hilo es la abstracción del sistema operativo de una actividad (el término deriva de la frase *hilo de ejecución – thread of execution*).

Un entorno de ejecución es la unidad de gestión de recursos: una colección de recursos gestionados por el *kernel* local a los que tienen acceso sus hilos. Un entorno de ejecución consta principalmente de:

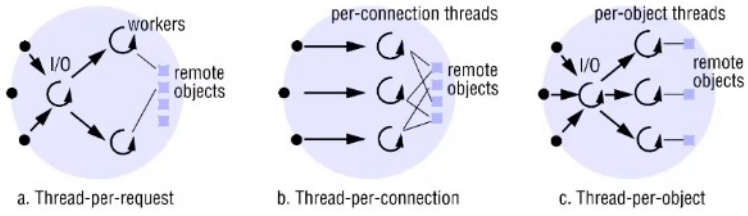
- Un espacio de direcciones.
- Sincronización de hilos – *threads* – y recursos de comunicación como semáforos e interfaces de comunicación (por ejemplo, sockets).
- Recursos de nivel superior, como archivos abiertos y ventanas.

Procesos e hilos
Processes and threads

En su forma más simple, al iniciar, el servidor crea un grupo fijo de hilos de trabajo para procesar las solicitudes. El módulo marcado como "recepción y puesta en cola – *receipt and queuing* –" generalmente se implementa mediante un único hilo encargado de recibir las solicitudes de un conjunto de sockets o puertos y las coloca en una cola compartida de solicitudes para que los trabajadores las recuperen.



Arquitecturas para servidores de múltiples hilos (*multi-threaded servers*)



a. Hilos por cada consulta(solo cuando le piden), cumple y se muere. **MEJOR POR SOLICITUD :D**

b. Hilo por usuario en conexión(no hay que hacer consulta), cumple y se muere. **CUESTAAAAAAAAA (Boom)**

c. Cada objeto tiene su propio hilo

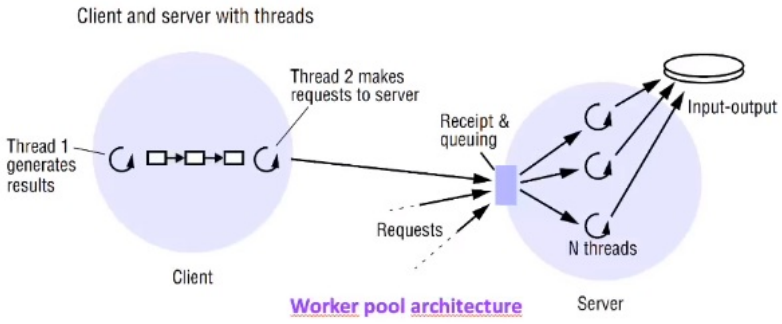
El entorno(Java) tiene procesos, cada proceso tiene acumulación de hilos(threads) o subprocesos.

Hilos = Procesos

Valos subprocesos = Procesos

Procesos e hilos
Processes and threads

La figura muestra una de las posibles arquitecturas de hilos – *threads* –, la arquitectura de colección de trabajadores – *worker pool architecture* –.



Hilos en java

Implementan prioridad y mecanismos de sincronización

Cualquier **clase** puede hacer hilo(Hilo padre(clase): clase general, saca clon(hijo), proceso, muere hijo)

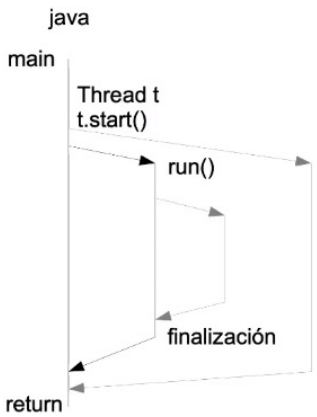
Clase recibe una solicitud(padre), clase tira objeto(hijo).

No es server, es SOFTWARE :D

Link [Concurrencia de Oracle]

Hilos

- Un hilo (Thread) es un proceso en ejecución dentro de un programa



- La finalización depende del hilo (`Thread.suspend, stop` están depreciados)
- Los hilos implementan prioridad y mecanismos de sincronización
- Cualquier clase se puede hacer hilo
  - `implements Runnable`

t instancia(objetos)

Hilos

```
public class PingPong extends Thread
{
 private String word;
 public PingPong(String s) {word=s;}

 public void run()
  {
   for (int i=0;i<3000;i++)
    {System.out.print(word);
     System.out.flush();}
   }

 public static void main(String[] args)
  {Thread tP=new PingPong("P");
   Thread tp=new PingPong("p");
   //tP.setPriority(Thread.MAX_PRIORITY);
   //tp.setPriority(Thread.MIN_PRIORITY);
   tp.start();
   tP.start();
   }
}
```

- Clase Thread
  - Implementa Runnable
- start() → run()
- ~~stop(), suspend()~~
- setPriority()
- sleep()
- Hereda de Object
  - wait(), notify()

Runnable interface debe ser implementado por cualquier clase

Hilo es una instancia(objeto) de una clase

start() --> run()

setPriority(): Importancia de ejecución

sleep(): Stand by (zombie)

Sincronización

- Los hilos se comunican generalmente a través de campos y los objetos que tienen esos campos
  - Es una forma de comunicación eficiente
  - Pero puede plantear errores de interferencias entre hilos
- La sincronización es la herramienta para evitar este tipo de problemas, definiendo órdenes estrictos de ejecución

campos = métodos con parámetros

Métodos sincronizados

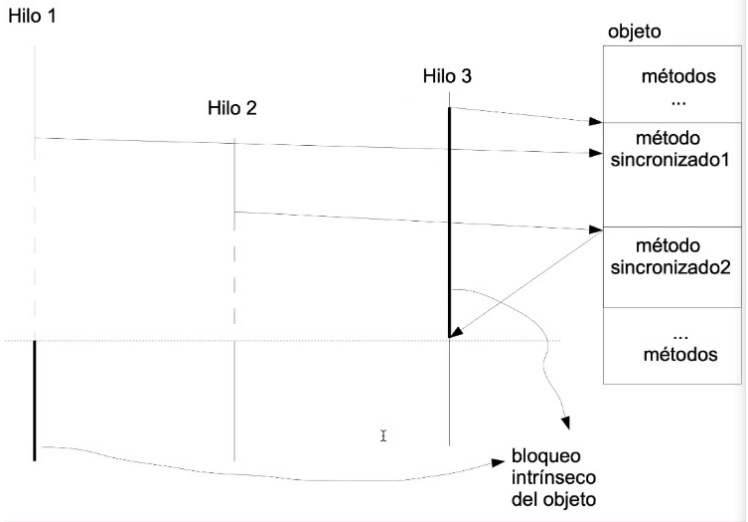
- Convertir un método en sincronizado tiene dos efectos:
  - Evita que dos invocaciones de métodos sincronizados del mismo objeto se mezclen. Cuando un hilo ejecuta un método sincronizado de un objeto, todos los hilos que invoquen métodos sincronizados del objeto se bloquearán hasta que el primer hilo termine con el objeto.
  - Al terminar un método sincronizado, se garantiza que todos los hilos verán los cambios realizados sobre el objeto.

No hace nada si no acaba el proceso anterior

Bloqueo intrínseco

- Cuando un hilo invoca un método sincronizado, adquiere el *bloqueo intrínseco* del objeto correspondiente.
- Si invoca un método *estático* sincronizado, adquiere el bloqueo intrínseco de la *clase*, independiente de los de sus objetos

Métodos sincronizados



Código sincronizado

- En vez de sincronizar todo un método podemos sincronizar una porción de código
  - Debemos especificar sobre qué quieren el bloqueo intrínseco

```
public void addName(String name)
   {
   synchronized(this)
      {
      lastName = name;
      nameCount++;
      }
   nameList.add(name);
   }
```

Se agrega synchronized

Problemas con múltiples procesos

- **Espera ocupada:** un proceso espera por un recurso, pero la espera consume CPU
  - `while(!recurso) ;` `wait()`
- **Interbloqueo** (deadlock): varios procesos compiten por los mismos recursos pero ninguno los consigue
- **Inanición**: un proceso nunca obtiene los recursos que solicita, aunque no esté interbloqueado
- **Autobloqueo**: un proceso espera por recursos que ya posee → sincronización reentrante

Espera ocupada: el zombie espera, aunque come memoria

Sincronización reentrante

- **Autobloqueo**: un proceso tiene el bloqueo intrínseco de un objeto
  - Mientras lo tiene, vuelve a pedirlo
- La sincronización reentrante evita que un proceso se bloquee a sí mismo en una situación de autobloqueo
  - Implementada en el núcleo de Java.

wait() y notify()

- **wait()** suspende el hilo hasta que se recibe una notificación del objeto sobre el que espera
  - Solución para *espera ocupada*
- El proceso que espera debe tener el bloqueo intrínseco del objeto que invoca al wait
  - Si no, da un error (IllegalMonitorStateException)
  - Una vez invocado, el proceso suspende su ejecución y libera el bloqueo
  - wait(int time) espera sólo durante un tiempo
- **notify()/notifyAll()** informan a uno/todos los procesos esperando por el objeto que lo invoca de que pueden continuar

EJEMPLO:

Sincronización

```
public class SynchronizedPingPong extends Thread
{
 private String word;
 public SynchronizedPingPong(String s) {word=s;}

 public void run()
  {
   synchronized(getClass())
    {
     for (int i=0;i<3000;i++)
      {
       System.out.print(word);
       System.out.flush();
       getClass().notifyAll();
       try
        {getClass().wait();}
       catch (java.lang.InterruptedException e) {}
      }
     getClass().notifyAll();
    }
  }

 public static void main(String[] args)
   {
   SynchronizedPingPong tP=new SynchronizedPingPong("P");
   SynchronizedPingPong tp=new SynchronizedPingPong("p");
   tp.start();
   tP.start();
   }
}
```

"Para entrar por aquí tenemos que conseguir el bloque intrínseco de la clase SynchronizedPingPong"

Ejecuto una iteración

Aviso de que he terminado

Espero un aviso

SINCRONO!!!!!!!

Conceptos

Principios solid: Buenas prácticas

Creacionales: Creación de clases

Estructurales: Creación de objetos

Comportamiento: Entre clases y objetos

Java no ELIMINA, Java QUITA la REFERENCIA

HEAP en Java

.flush():

Referencias

Coulouris, G. (2012). DISTRIBUTED SYSTEMS Concepts and Design. Boston MA: Pearson Education, Inc.

Donahoo, M. (2009). TCP/IP SOCKETS IN C Practical guide for programmers. Burlington, MA: Morgan Kaufmann.

IBM. (22 de 07 de 2023). IBM Documentation. Obtenido de Socket characteristics: https://www.ibm.com/docs/en/i/7.2?topic=programming-socket-characteristics

TAREA

Carrera 4x100

- Implementar una carrera por relevos, similarmente al ejercicio anterior:
  - Tenemos 4 `Atletas` dispuestos a correr
  - Tenemos una clase principal `Carrera`
  - Tenemos un objeto estático `testigo`
  - Todos los atletas empiezan parados, uno comienza a correr (tarda entre 9 y 11s) y termina su carrera e inmediatamente comienza otro
  - Pistas:
    - Thread.sleep y Math.random para la carrera
    - synchronized, wait y notify para el paso de testigos
    - System.currentTimeMillis o Calendar para ver tiempos