Programación Concurrente

Curso Java 2012, 2013

Juan Manuel Fernández Peña

Programación secuencial

- Hasta ahora se ha trabajado con programas secuenciales
  - Se ejecuta paso a paso
  - Son deterministas: si se ejecutan varias veces en las mismas condiciones, en diferentes equipos, da los mismos resultados

¿Cómo realizaría …?

- Tiene un robot en Marte, buscando muestras del suelo, que debe analizar; periódicamente debe enviar datos a Tierra. Cada cierto tiempo el robot no está alineado correctamente con Tierra durante un tiempo.
- Si tuviera un solo procesador central en Tierra, ¿cómo haría para que funcionara el sistema?
- ¿Qué alternativas se le ocurren?

¿Cómo realizaría …?

- Tiene un banco pequeño con diez cajeros automáticos conectados vía telefónica (cable dedicado, exclusivo). Si utiliza únicamente el procesador en la computadora del banco, ¿cómo debería ser la programación para atender todo? ¿qué efectos tendría que adquieran dos nuevos cajeros?
- ¿qué alternativas se le ocurren?

¿Cómo realizaría …?

- Tiene una computadora personal como las de la escuela o su laptop. ¿Cómo hacen los programas para tener Chrome y estar viendo noticias, usando un servicio de chat o Tweeter, mientras revisa el material del curso en un archivo pdf y va representando elementos en una tabla de Excel?

¿Cómo realizaría …?

- Tiene una matriz de datos enorme (10 millones de datos) y debe aplicar una función a cada dato. ¿cómo lo hace?

- Ahora, se da cuenta que su procesador tiene dos núcleos (dos procesadores); ¿cómo lo aprovecharía?

¿Cómo realizaría …?

- Debe hacer un juego de video como Packman, donde hay un personaje manejado por el usuario humano y uno o más personajes manejados por el programa; ¿cómo haría su programación? ¿podría aprovechar los conceptos OO? ¿qué más le haría falta?

- Un juego como Tetris, donde hay piezas que se mueven mientras se atiende algún dispositivo de entrada; ¿cómo haría su programación? ¿podría aprovechar los conceptos OO? ¿qué más le haría falta?

Programación Concurrente

- En lo que sigue se trata de programación concurrente
  - En cada momento puede haber varios fragmentos en ejecución ***más o menos simultánea***
  - Dos ejecuciones, en las mismas circunstancias, ***pueden*** producir resultados diferentes

Concurrencia

De diccionario

- Concurrencia:
  - simultaneidad de dos sucesos
  - galicismo por competencia, rivalidad

Computacional

- Dos o más tareas son concurrentes si el inicio de una de ellas ocurre entre el inicio y el final de otra

Concurrencia en la vida real

- Varias personas participando en actividad colectiva: fiesta, reunión, viajando juntos
- Las actividades de una empresa, donde hay varias personas y máquinas trabajando en un mismo período de tiempo
- Una persona que está atendiendo su trabajo y, a la vez, responde teléfono, consultas por Internet, oye música y toma un café

Concurrencia en la vida real

- Se presentan situaciones de competencia por los recursos, incluso a nivel de lucha por ellos
- Puede aprovecharse para colaborar en realizar una tarea compleja, dividiéndola en tareas menores
- Surgen problemas de comunicación y de sincronización

Concurrencia en computación

- Procesador mucho más rápido que otras unidades de la computadora y estas más que los seres humanos que la usan
- Desde hace tiempo se usa la multiprogramación para compartir el tiempo de un procesador entre varios trabajos
- Algunas computadoras tienen varios procesadores que permiten un paralelismo mayor

Concurrencia en computación

- Computadoras personales permiten varios programas ejecutándose simultáneamente:
  - Sistema operativo
  - Antivirus
  - Internet: navegador, varios sitios
  - Chat
  - Procesador de textos
  - Juegos

Concurrencia en computación

- Un mismo programa permite simultáneamente:
  - Varias ventanas activas
  - Varios elementos gráficos activos en una ventana (botones, etc.)
  - Entrada por teclado
  - Entrada por ratón
  - Entrada por otros dispositivos
  - Uso de impresora

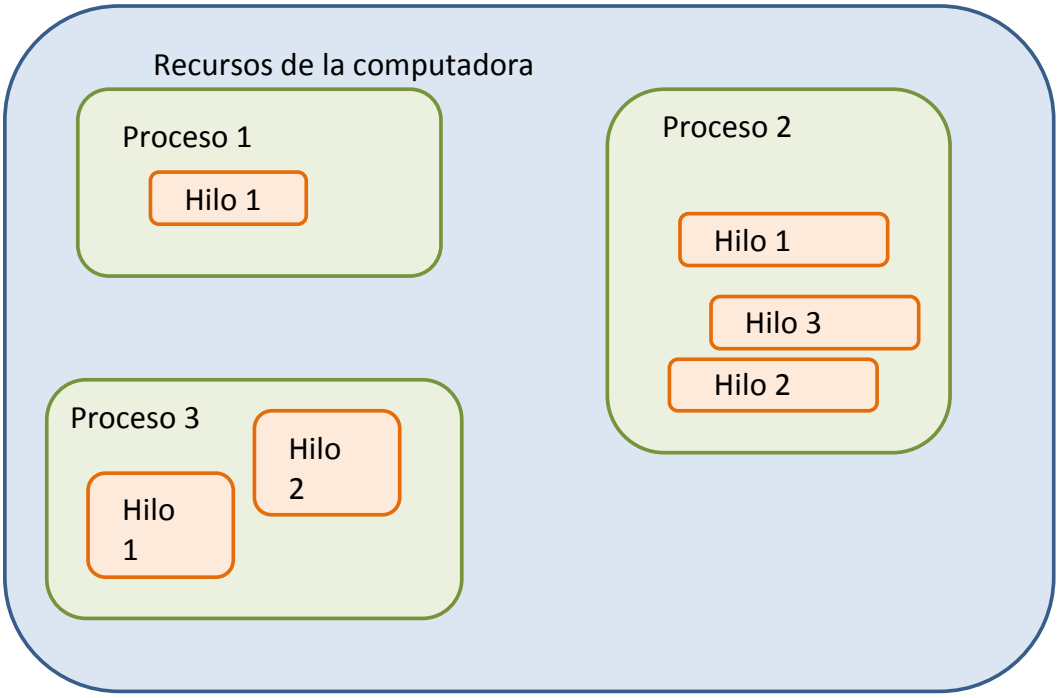
Proceso

- “Programa al tiempo de ejecución”
- “Abstracción del sistema operativo que permite que un sistema informático soporte múltiples unidades de ejecución”
- De acuerdo al modelo de memoria simple, un programa aceptado para ejecución recibe una serie de recursos:
  - contador,
  - área de almacenamiento de objetos,
  - área de métodos,
  - contexto (pila, contadores auxiliares, etc.)

Hilo (Thread, Hebra)

- Proceso ligero, que no posee recursos propios, sino que los comparte con otros
- “Secuencia de llamadas que se ejecutan independientemente de otras, mientras que, posiblemente al mismo tiempo, comparte recursos tales como archivos y memoria, además de acceder a otros objetos del mismo programa”
- Un proceso tiene al menos un hilo de ejecución
- Los procesos que emplean interfaces gráficas tienen varios hilos

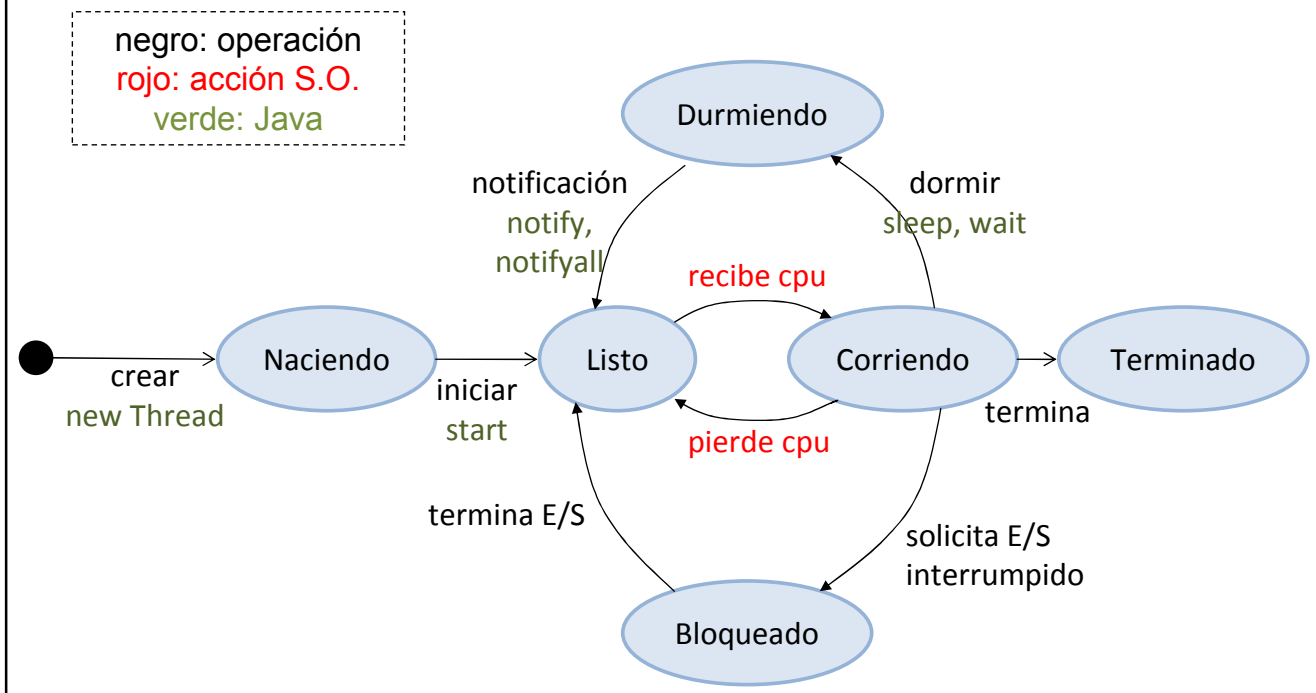
Procesos e Hilos



Comunicación

- Entre **Procesos**: requiere apoyo del sistema operativo o el uso de recursos externos al proceso
- Entre **Hilos**: a cargo del programa, usando estructuras de datos

Vida de un hilo



Problemas potenciales

- Nunca pasa a “Corriendo”
- Se queda bloqueado o durmiendo
- Seguridad vs Vivacidad …

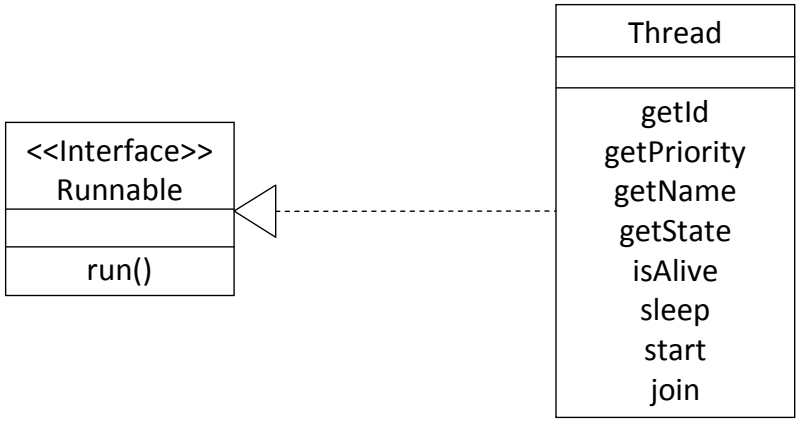
Impredecibilidad de resultados

- Analice la ejecución de ejemplo de generación de primos
  - Cada corrida produce resultados ligeramente diferentes
- Analice la ejecución del ejemplo de mensaje A,B,1,2
  - Resultados iguales en unas máquinas y variables en otras

Hilos en Java

- Un hilo principal que inicia en un método **main**
- Hilos asociados a elementos gráficos
- Hilos específicos creados por el desarrollador

Hilos en Java



Nota: Thread implementa la interfaz Runnable

Hilos en Java

- Como clase:

```
public class Unhilo extends Thread{
. . .
public void run(){
    // función del hilo; puede ser un ciclo
}
```

- Como interfaz que se implementa en otras clases:

```
public class Otrohilo extends Algo implements Runnable{
. . .
public void run(){
 // función del hilo; puede ser un ciclo
}
```

Hilos en Java
Funcionamiento del método run

- El contenido del método **run** indica todo lo que hace el hilo en su vida
  - Secuencia de acciones, con posibles descansos intercalados (ejemplo A B 1 2)
  - Ciclo dentro de un for (ejemplo primos)
  - Ciclo controlado por una variable (volátil) que sirve para pararlo cuando se cumple una condición interna o externa
- Puede incluir sleep, wait, etc.

Dormir, ¿para qué?

- Cuando un hilo no tiene E/S se apodera del procesador y no permite concurrencia
- El sleep obliga a soltar el procesador
- Unidad: milisegundo
- Uso de sleep

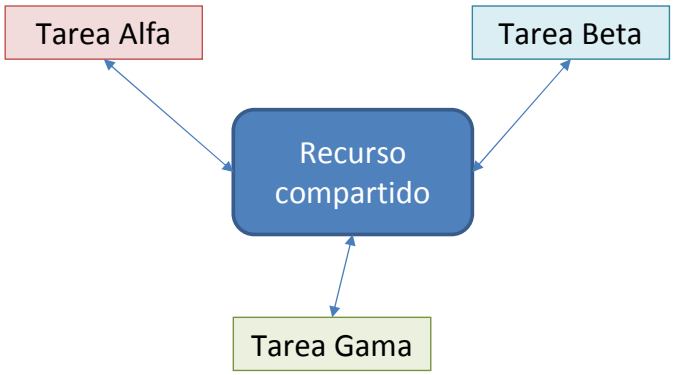
```
try{ sleep(miliseg);} catch (InterruptedException ie)
{ // lo que se hace si falla}
```

- El **main** tiene un hilo que puede dormir
- **sleep** es método estático, se puede invocar de modo genérico: Thread.sleep(miliseg)

Problemas de concurrencia

- Compartir recursos
  - Cuando se comparte, hay riegos de abuso:
    - Alguien se apodera de recurso
    - Se toma como bueno un dato obsoleto
    - Se borra un dato antes de ser utilizado
- Sincronización
  - Tareas que deben esperar el fin de otras
  - Tareas que deben esperar una condición

Recursos compartidos



Algunos conceptos

- Región crítica o sección crítica: segmento de código que corresponde al uso de un recurso compartido.
- Exclusión mutua: solución para evitar conflictos en una región crítica; sólo se permite una tarea en ella.
- Condición de carrera: cuando una tarea se adelanta indebidamente en el uso de una región crítica (lee antes de que haya datos o escribe antes de que los hayan usado)

Variables volátiles

- Las operaciones de acceso a variables compartidas en memoria pueden llegar a ser interrumpidas, permitiendo que se utilice un valor desactualizado o se pierda otro; problemas de consistencia.
- Se busca realizar operaciones atómicas, que no puedan ser interrumpidas.
- Op. Atómica: leer o escribir una variable, excepto long o double (en sistemas de 32 bits)
- Op. Atómica: indicar que es volátil
  - long volatile variable;

synchronized

- Otra solución para operaciones atómicas más complejas:
  - Synchronized aplicado a métodos

```
public void synchronize deposita(int n){
    saldo += n;
    tt++;
    }
```

  - Un método synchronized opera como sigue: **todo el objeto queda bloqueado**, de modo que otras llamadas a ese método o a otro del objeto deben esperar (sólo los que digan “synchronized”)
  - Funciona como una llave de un cuarto compartido: el que llega la toma y la devuelve al salir

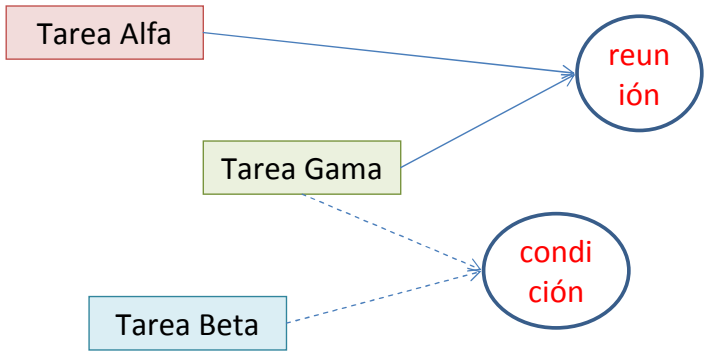
Comparación

Técnica	Ventajas	Desventajas
synchronized	La memoria privada se concilia con la principal cuando se obtiene la llave y cuando se regresa.	Elimina concurrencia; produce bloqueo, reduciendo vivacidad.
volatile	Permite concurrencia.	La memoria privada se concilia con la principal con cada acceso.

Recomendaciones dentro de synchronized

- Mantener código pequeño
- Evitar instrucciones que pueden ser interrumpidas (E/S)
- No invocar métodos en otros objetos

Sincronización



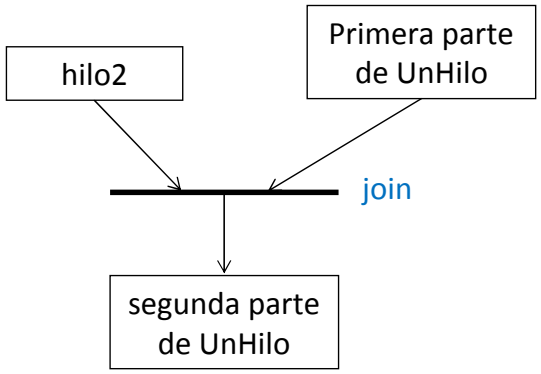
Sincronización

- Sincronizar tareas es ponerlas de acuerdo cuándo pueden hacer cosas
  - Cuándo una termina
  - Cuándo se cumple una condición

Espera a que termine

- Una tarea no prosigue hasta que otra termina
- Método join()

```
Class UnHilo{ …
  public void algo(){…
     hilo2.join(); //hilo2 es otro hilo diferente
     …}
}
```



Espera por una condición

- Para que una tarea espere:
  - Puede dormir, pero puede desperdiciar tiempo
  - Puede hacer un ciclo, pero sigue usando el procesador
  - Java: ciclo con condición y wait; no usa procesador, lo despiertan con notify o notifyall

Espera tradicional

- En un método, llamado por un hilo:

```
while (noSigas)
   { // ciclo inútil
   }
```

Mientras espera, no suelta el procesador

- En otro método, llamado por otro hilo:

```
noSigas = true;
//trabajo exclusivo
noSigas = false; //le permite al otro trabajar
```

Mientras no tenga el procesador, no puede dejar pasar a la otra

Espera sin usar procesador

- En un método, llamado por un hilo :

```
while (noSigas)
   { // ciclo inútil
       wait();
   }
```

Mientras espera, libera el procesador

- En otro método, llamado por otro hilo:

```
noSigas = true;
//trabajo exclusivo
notifyAll();
noSigas = false; //le permite al otro trabajar
```

Avisa a los que están esperando que pueden seguir; si hay varios, sólo uno tomará el procesador

Espera sin usar procesador

- En un método, llamado por un hilo :

```
while (noSigas)
   { // ciclo inútil
       wait();
   }
```


- En otro método, llamado por otro hilo:

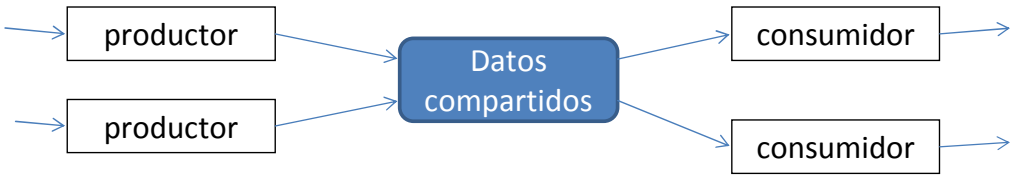
```
noSigas = true;
//trabajo exclusivo
notify();
noSigas = false; //le permite al otro trabajar
```

Notas en wait y notify

- Todos los objetos los tienen
- Debe usarse en un objeto común (llamado a veces monitor)
- Deben usarse dentro de métodos synchronized del objeto común
- Puede haber *wait* en varios monitores a la vez

Ejemplo clásico: productores y consumidores

- En la realización de un trabajo complejo colaboran varias tareas; unas proporcionan datos a una estructura de datos común (productores), mientras otras utilizan los datos de la estructura para realizar diversas acciones (consumidores)



Objeto común, usado como monitor (jagan)

```
class Q {
      int n;
      boolean valueSet = false;
      synchronized int get() {
        if(!valueSet)
        try {
          wait();
        } catch(InterruptedException e) { System.out.println("InterruptedException
            caught");}

      System.out.println("Got: " + n);
      valueSet = false;
      notify();
      return n;
      }
```

Objeto común, usado como monitor

```
    synchronized void put(int n) {
      if(valueSet)
        try {
          wait();
        } catch(InterruptedException e) { System.out.println("InterruptedException
              caught"); }
      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
    }
}
```

Productor

```
class Producer implements Runnable {
    Q  q;
    Producer(Q  q) {
    this.q = q;
    new Thread(this, "Producer").start();
    }
    public void run() {
       int i = 0;
       while(true) {
          q.put(i++);
       }
    }
}
```

Consumidor

```
class Consumer implements Runnable {
    Q  q;
    Consumer(Q q) {
      this.q = q;
      new Thread(this, "Consumer").start();
      }
    public void run() {
      while(true) {
         q.get();
      }
    }
}
```

Principal

```
class PCFixed {
   public static void main(String args[]) {
      Q  q = new Q();
      new Producer(q);
      new Consumer(q);
      System.out.println("Press Control-C to
   stop.");
   }
}
```

Uso de mecanismos

Para resolver problemas de concurrencia

- Identifique sus objetivos, que sean claros
- Identifique la sección crítica (recurso compartido, segmento de código sonde se usa el recurso compartido)
- Utilice algún mecanismo

Elementos de mecanismo

Estado (variable o variables que indican el estado. Generalmente hay dos estados: Abierto (se permite acceso a región crítica) o Cerrado (no se puede acceder a región crítica)

- Puede ser una simple variable booleana o un número entero
- Van dentro de una clase que será compartida
- Se usan a través de métodos sincronizados

Elementos de mecanismo

Condición o condiciones de acceso a la región crítica

- Debe ser algo ligado al estado
- Generalmente dentro de un ciclo
- Debe evitarse un “falso despertar”

```
while (condición) wait();
```

Elementos del mecanismo

Cambio en la variable de estado

- No basta considerar la condición, debe haber un cambio

Elementos del mecanismo

Notificación

- En casi todos los casos se avisa a los hilos que están esperando para usar la región crítica
- Puede usarse
  - notify() avisa a uno al azar.
  - notifyAll() notifica a todos
  - hilo.notify() avisa a un hilo específico

Mecanismos: Cerrojo

```
public class Cerrojo{
  private boolean protegido = false;
  public synchronized void abrir() throws InterruptedException{
    while (protegido)    wait();
    protegido = true;

  }
  public synchronized void cierra(){
    protegido = false;
    notify();
  }
}
```

Cerrojo

- Para usar cerrojo:
  - Todos los hilos que comparten la región crítica reciben copia de la referencia al cerrojo
  - El hilo que necesita usar la región crítica invoca “abrir”, luego realiza las acciones que deba y finalmente invoca “cierra”.
  - Las acciones protegidas no deben incluir uso de archivos ni otras acciones que puedan originar retrasos considerables o espera infinita.

Mecanismo: Lectores y escritores

- Supone que existen dos tipos de hilo que comparten la región crítica (RC): unos que solo la consultan (lectores) y otros que la modifican (escritores).
- En cualquier momento solo puede haber un Escritor usando la RC
- En cualquier momento puede haber varios Lectores consultando la RC
- Para los hilos que esperan, tiene mayor precedencia un Escritor

Lectores y Escritores

```
Public class ReadWrite{
  private int lectores = 0;
  private int escritores = 0;
  private int solicitanescribir = 0;
  public synchronized void entraLector throws InterruptedException{
    while (escritores>0 || solicitanescribir >0)
      wait();
    lectores++;
  }
  public synchronized void saleLector(){
    lectores--;
    notifyAll();
  }
```

Lectores y Escritores

```
  public synchronized void entraEscritor() throws InterruptedException{
    solicitanescribir++;
    while (lectores>0 || escritores>0)      wait();
    solicitanescribir--;
    escritores++;
  }
  public synchronized void saleEscritor(){
    escritores--;
    notifyAll();
  }
}
```

Mecanismo: semáforo

- Con la idea de un semáforo.
- Existe un número de permisos (1 o más)
- Mientras haya permiso, un hilo puede acceder a RC, si no, debe esperar
- Dos usos:
  - Como protección de la RC, semejante al cerrojo
  - Como señal para sincronizar dos hilos (deben compartir el semáforo):
    - Hilo Generador hace algo y luego invoca “toma”
    - Hilo Receptor invoca “libera” y luego hace su tarea

Semáforo

```
Public class Semáforo{ // es un semáforo contador)
  private boolean señales = 0;
  private int límite = 0;
  public Semáforo(int lim){   límite = lim;   }
  public synchronized void toma() throws InterruptedException{
    while (señales == límite)     wait();
    señales++;
    notify();
  }
  public synchronized void libera() throws InterruptedException{
    while (señales == 0)   wait();
    señal--;
    notify();
  }
}
```

Mecanismo: Cola con bloqueo

- Útil cuando dos tipos de hilos colaboran, uno enviando mensajes y otro atendiéndolos.
- Cuando la cola tiene lugar, se deja el mensaje, pero si no hay lugar se espera
- Si se va a retirar un mensaje y existe, se retira; si no hay, debe esperar
- Sólo en los límites se bloquea algún hilo
- Note similitud con semáforo.

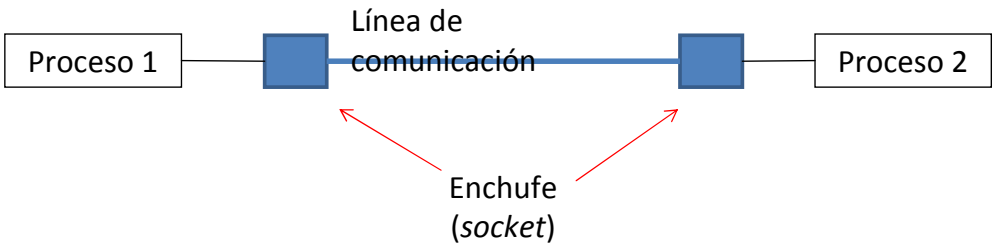
Cola con bloqueo

```
Public class ColaBloqueo{
  private List cola = new LinkedList();
  private int límite = 10;
  public ColaBloqueo(int lim) {   límite = lim;   }
  public synchronized void agrega(Object item)
        throws InterruptedException{
    while (cola.size() == límite)     wait();
    if (cola.size() == 0) notifyAll();
    cola.add(item);
  }
Public synchronized Object saca() throws InterruptedException{
    while (cola.size() == 0)     wait();
    if (cola.size() == límite)   notifyAll();
    return cola.remove(0);
  }
}
```

Comunicación entre procesos

Conceptos generales

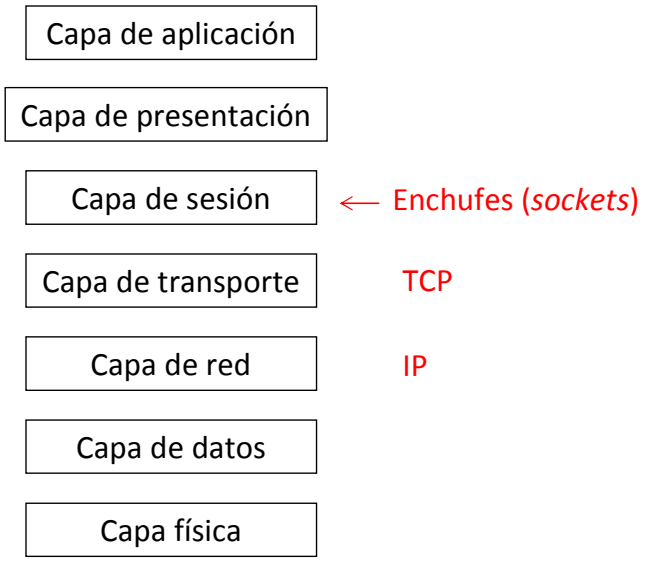
- Para comunicar dos procesos (sin recursos comunes) se debe establecer una línea de comunicación.
- Los extremos de la línea se conocen como enchufes (*sockets*).
- Cada enchufe tiene una dirección (url) y un puerto



Conceptos generales

- La línea puede ser de muchos tipos: enlace interno a un archivo, comunicación con http, etc.
- Los enchufes pueden estar en una misma máquina o en dos diferentes.
- El url (*universal resource locator*) es un tipo de dirección que permite identificar recursos en cualquier lugar de una red. Permite localizar la computadora a la cual se dirige una petición. *Localhost* para misma máquina.
- El puerto es un número de contacto, permite operar varios canales de comunicación entre computadoras. Cada aplicación que debe atender comunicación se conecta a uno; como un número de departamento. Permite asociar aplicación.
- Puertos entre 0 y 65535; protegidos hasta 1023; muchos apartados: 7: eco; 20 y 21: FTP, 25: SMTP (correo); 53: DNS; 130: CISCO; 1433: Microsoft SQL Server

Relación con modelo de capas



Sockets parten datos en paquetes que viajan por la capa de transporte y los entregan en la capa de sesión

Tipos

- Enchufes TCP (transmission control protocol):
  - Establecen una sesión entre los extremos, permitiendo una comunicación como archivos de texto (streams); cuidan el orden de sus paquetes y su reenvío si se pierden. Como llamar por teléfono.
- Enchufes UDP (*user datagram protocol*):
  - No establecen sesión, son como un envío de carta o telegrama; pueden llegar bien o parcialmente; pueden perder orden. Son más rápidos.

Ver ejemplos en archivos aparte

Enchufes TCP en Java

- El paquete java.net incluye las clases
  - URLConnection: permite leer o escribir en un url; usa enchufes sin mostrarlos
  - Socket: enchufe cliente (manda solicitud)
  - ServerSocket: enchufe servidor (atiende solicitud)

Enchufe UDP en Java

- Del paquete java.net
  - DatagramSocket: enchufe para envío y recepción de paquetes que forman datagramas; para envío especifica puerto destino
  - DatagramPacket: paquete de datos; para servidor especifica puerto donde escucha

Clientes múltiples

- Un servidor puede atender clientes múltiples (default 50), pero
  - Si lo hace secuencial, atiende uno hasta terminar; los demás esperan, o bien
  - Por cada cliente que le llega genera un hilo para atenderlo, evitando que esperen en cola
- En general el enchufe servidor debe ser lo más independiente del resto, para evitar bloqueos; use un hilo

Ejemplo integrado

- Un banco ofrece un cajero interactivo y además recibe peticiones remotas vía un enchufe. (Ver código en archivo aparte)