Ejemplo: Accesos a un recurso compartido con sincronización

Ejemplo que vamos a ver

Vamos a modificar las dos aplicaciones del ejemplo anterior:

1. Aplicación1, accede a un fichero que contiene un valor entero. Lee ese valor, lo incrementa en 1 y escribe ese valor actualizado en el mismo fichero.
   - **Identificaremos** las **instrucciones que formarán la sección o secciones críticas** en la anterior aplicación.
   - **Incluiremos mecanismos de sincronización** (proporcionados por el sistema operativo).
2. Aplicación2, crea varios procesos de la Aplicación1. De forma que todos esos procesos utilizarán el mismo fichero.

Veremos cómo probar el ejemplo. ¿El resultado final, es el esperado? ¿Cómo vemos qué es lo que ha pasado?

Para la implementación de estos ejemplos, hemos utilizado NetBeans.
y proyectos estándar de consola Java.

Aplicación 1: Modificar, para incluir sincronización.

1. ¿Qué **instrucciones** de nuestra aplicación **deben ejecutarse como una unidad indivisible**?
   1. **Leer** el valor del fichero.
   2. **Incrementa** el valor (modificarlo).
   3. **Escribir** ese valor en el fichero del que se había leído.

   Estas van a ser las **instrucciones de nuestra sección crítica**.

2. **Sólo un proceso entrará a ejecutar las instrucciones de su sección crítica**. **El resto** de procesos, **encontrarán el recurso bloqueado y esperarán** hasta que el que consiguió entrar finalice y lo desbloquee. Para ello utilizaremos un objeto de tipo **FileLock**.

3. Necesitamos poder **abrir el fichero con permisos de lectura y escritura**. Utilizaremos un objeto de la clase **RandomAccessFile**. Así no utilizaremos un objeto para leer (FileReader) y otro para escribir (FileWriter).

4. Por supuesto, continuaremos **gestionando las posibles excepciones**, tal y como hemos aprendido en el módulo Programación del primer curso.

Aplicación 1: Modificar el valor de un fichero.

Comencemos:

- **Abrimos el fichero en modo rwd** (lectura-escritura y las escrituras serán llevadas directamente a disco).
- Resaltado en rojo, las **instrucciones que forman la región crítica**:
  - **Solicitamos el bloqueo** del canal de acceso al fichero.
  - **Leer** valor de fichero.
  - **Incrementar** valor.
  - **Escribir** valor.
  - **Liberar el bloqueo** del fichero.
- La **solicitud y liberación del bloqueo son las instrucciones** que marcan el inicio y el fin de la región crítica. De hecho, son las instrucciones **que crean la región crítica**.

```java
//Preparamos el acceso al fichero
archivo = new File(nombreFichero);
for (int i=0; i<100; i++)//aumentamos las situaciones de concurrencia
   try{
      raf = new RandomAccessFile(archivo,"rwd"); //Abrimos el fichero
      //***************
      //Sección crítica
      bloqueo = raf.getChannel().lock();
      //bloqueamos el canal de acceso al fichero. Obtenemos el objeto que
      //representa el bloqueo para después poder liberarlo
      System.out.println("Proceso"+ orden +
            ": ENTRA sección");
      // Lectura del fichero
      valor = raf.readInt(); //leemos el valor
      valor ++; //incrementamos
      raf.seek(0); //volvemos a colocarnos al principio del fichero
      raf.writeInt(valor); //escribimos el valor
      System.out.println("Proceso"+ orden +
            ": SALE sección");
      bloqueo.release(); //Liberamos el bloqueo del canal del fichero
      bloqueo = null;
      //Fin sección crítica
      //*******************
      System.out.println("Proceso"+ orden +
            ": valor escrito " + valor);
  }catch(Exception e){
      System.err.println("P"+orden+" Error al acceder al fichero");
      System.err.println(e.toString());
  }finally{
      try{
         if( null != raf ) raf.close();
         if( null != bloqueo) bloqueo.release();
      }catch (Exception e2){
         System.err.println("P"+orden+" Error al cerrar el fichero");
         System.err.println(e2.toString());
         System.exit(1);  //Si hay error, finalizamos
      }
  }
```

Aplicación 1:
Instrucciones de la región crítica.

Revisando las instrucciones de nuestra región crítica:

```
//Sección crítica
bloqueo = raf.getChannel().lock();
//bloqueamos el canal de acceso al fichero. Obtenemos el objeto que
//representa el bloqueo para después poder liberarlo
System.out.println("Proceso"+ orden +
      ": ENTRA sección");
// Lectura del fichero
valor = raf.readInt(); //leemos el valor
valor ++; //incrementamos
raf.seek(0); //volvemos a colocarnos al principio del fichero
raf.writeInt(valor); //escribimos el valor
System.out.println("Proceso"+ orden +
      ": SALE sección");
bloqueo.release(); //Liberamos el bloqueo del canal del fichero
//Fin sección crítica
```

- La **región crítica creada, sólo protege al recurso**. Sólo protege los accesos al fichero y asegura que mientras que este proceso tiene el fichero bloqueado, ningún otro proceso accederá al fichero. Como el bloqueo no se libera entre la lectura y la escritura, otro proceso no podrá leer el mismo valor que haya leído este proceso.
- El resto de instrucciones que están entre el bloqueo y desbloqueo del canal no están protegidas. Es por esto, que en el fichero con las salidas de los procesos (javalog.txt), pueden mezclarse los textos de salida de unos procesos con otros.
- En la siguiente unidad (Hilos), veremos cómo crear **secciones críticas** que protegen todas las instrucciones que las contienen, ya que se crean **a nivel de instrucciones** y no **a nivel de recurso**. Esto será así, porque **los hilos de un mismo proceso pueden compartir variables** (y zonas de memoria). Las variables no las protege el sistema operativo como sucede con los ficheros. **El lenguaje de programación nos proporcionará los mecanismos para crear regiones críticas a nivel de grupo de instrucciones**; y todas ellas serán ejecutadas de forma atómica y exclusiva.
  La **protección a nivel de recurso proporcionada por el sistema operativo, siempre la tendremos disponible** (si el lenguaje y el SO la tienen implementada); tanto para procesos, como para hilos de un proceso.

Aplicación 1:
Método FileChannel.lock()

- El método **lock()** de la **clase FileChannel,** nos devuelve un objeto de tipo **FileLock** que representa al bloqueo que ha obtenido el proceso sobre el canal de acceso al fichero.
- Para el sistema, el **fichero aparecerá como abierto en uso exclusivo** para ese proceso.
- Sólo podremos solicitar el **bloqueo** de un fichero, si lo hemos **abierto con permisos de escritura**.
- El método **lock()**, **suspende al proceso** hasta que esté disponible el bloqueo.
- El método lock() puede **generar una excepción**.
- En el caso de nuestro ejemplo, si no se produce ninguna excepción, el proceso incrementará 100 veces el valor del fichero. Si sucede alguna excepción, se notificará el error y se continuará intentando el resto de incrementos.
- Date cuenta de que la sección crítica, está definida dentro del bucle y abarca las instrucciones que acceden al recurso compartido. Y, no, está el bucle completo dentro de la región crítica.
- No protegemos las escrituras de salida, porque nos da igual las condiciones que de competencia que se puedan presentar.

```
//Preparamos el acceso al fichero
archivo = new File(nombreFichero);
for (int i=0; i<100; i++)//aumentamos las situaciones de concurrencia
   try{
      raf = new RandomAccessFile(archivo,"rwd"); //Abrimos el fichero
      //***************
      //Sección crítica
      bloqueo = raf.getChannel().lock();
      //bloqueamos el canal de acceso al fichero. Obtenemos el objeto que
      //representa el bloqueo para después poder liberarlo
      System.out.println("Proceso"+ orden +
           ": ENTRA sección");
      // Lectura del fichero
      valor = raf.readInt(); //leemos el valor
      valor ++; //incrementamos
      raf.seek(0); //volvemos a colocarnos al principio del fichero
      raf.writeInt(valor); //escribimos el valor
      System.out.println("Proceso"+ orden +
           ": SALE sección");
      bloqueo.release(); //Liberamos el bloqueo del canal del fichero
      bloqueo = null;
      //Fin sección crítica
      //*******************
      System.out.println("Proceso"+ orden +
           ": valor escrito " + valor);
  }catch(Exception e){
      System.err.println("P"+orden+" Error al acceder al fichero");
      System.err.println(e.toString());
  }finally{
      try{
         if( null != raf ) raf.close();
         if( null != bloqueo) bloqueo.release();
      }catch (Exception e2){
         System.err.println("P"+orden+" Error al cerrar el fichero");
         System.err.println(e2.toString());
         System.exit(1);  //Si hay error, finalizamos
      }
  }
```

Aplicación 1:
Método FileChannel.tryLock() y la espera activa.

- El método **FileChannel.tryLock(), funciona igual que el método lock(),** salvo porque, **no bloquea al proceso**. También genera excepciones.
- Por supuesto, **siempre** es más **recomendable sustituir métodos que puedan generar excepciones por sus equivalentes que no las generen**.
- Pero debemos **tener mucho cuidado con no crear en una espera activa** que pueda bloquear el sistema.

```
// Ejemplo de espera activa
while ((bloqueo = raf.getChannel().tryLock()) != null);
```

Un bucle sin instrucciones en su cuerpo, que sólo chequea su condición.

- **Espera activa**: se produce cuando un proceso, **continuamente testea el estado de un recurso o variable**. Lo que hace es **consumir tiempo de CPU en una tarea poco productiva**.
  - ¿Cómo **evitar** los **problemas** que puede provocar?
    A) **Sustituir la espera activa por métodos bloqueantes equivalentes**. Será el sistema el encargado de suspender el proceso hasta que el recurso esté disponible.
    B) **Limitar el número de intentos**, que no sean infinitos. Por ejemplo, cuando intentamos acceder a una página web y el servidor está caído, nuestro navegador, realiza varios intentos esperando un tiempo entre ellos; después de ese número de intentos, nos notifica que no ha podido conectar y que podemos intentarlo después. Nosotros deberemos hacer lo mismo.
- En nuestro caso, como disponemos del método **FileChannel.lock() que suspende al proceso hasta que le sea concedido el bloqueo del fichero (caso A)**, no nos plantearnos utilizar el método FileChannel.tryLock().
- Utilizaremos **FileChannel.tryLock()** en los casos en los que queramos **consultar** si un fichero se puede o no bloquear (o si ya está bloqueado), pero **no para obtener el bloqueo del recurso**.

Aplicación 2: Lanzar la ejecución de varias instancias de la Aplicación 1.

- En este ejemplo, vamos a incluir la **creación inicial del fichero en la Aplicación2** y no en la Aplicación1 como hicimos anteriormente.

```
archivo = new File(nombreFichero);
//Preparamos el acceso al fichero
if (!archivo.exists()){
  //Si no existe el fichero
  try {
      archivo.createNewFile(); //Lo creamos
      raf = new RandomAccessFile(archivo,"rw"); //Abrimos el fichero
      //El modo tiene que ser lectura-escritura. No es posible sólo escritura
      raf.writeInt(0); //Escribimos el valor inicial 0
      System.out.println("Creado el fichero.");
  }catch(Exception e){
      System.err.println("Error al crear el fichero");
      System.err.println(e.toString());
  }finally{
      try{ // Nos asegurarnos que se cierra el fichero.
          if (null != raf)
              raf.close();
      } catch (Exception e2) {
              System.err.println("Error al cerrar el fichero");
              System.err.println(e2.toString());
              System.exit(1); //Si hay error, finalizamos
      }
  }
 }
```

Aplicación 2: Lanzar la ejecución de varias instancias de la Aplicación 1.

- La creación de varias instancias de la Aplicación 1 es igual que hemos hecho hasta ahora.

```
//Creamos un grupo de procesos que accederán al mismo fichero
try{
     for (int i = 0; i <=25; i++){
         nuevoProceso = Runtime.getRuntime().exec("java -jar "+
             "AccesoMultipleFichero.jar " + i + " "+ args[0]);
         //Creamos el nuevo proceso y le indicamos el número de orden y
         //el fichero que debe utilizar.
         System.out.println("Creado el proceso " + i);
         //Mostramos en consola que hemos creado otro proceso
     }
 }catch (SecurityException ex){
     System.err.println("Ha ocurrido un error de Seguridad."+
             "No se ha podido crear el proceso por falta de permisos.");
 }catch (Exception ex){
     System.err.println("Ha ocurrido un error, descripción: "+
             ex.toString());
 }
```

Podemos apreciar, en el comando con el que lanzamos la ejecución del proceso, que hemos incluido paso de parámetros por la línea de comandos: número de orden del proceso y ruta del fichero a utilizar.

Aplicación 1:
Incluir código para tratar los argumentos recibidos en la línea de comandos.

Como hemos visto en el código anterior, la aplicación 2 pasará parámetros por línea de comandos a los procesos de la aplicación 1. Hemos decidido que los procesos reciban los siguientes parámetros:

1. El número de orden de creación de ese proceso.
2. El nombre o ruta del archivo que queremos que utilice.

Con estos dos parámetros, buscamos poder ver de forma más clara qué sucede durante la ejecución de los procesos, y poder cambiar el fichero al que acceden los ficheros, para hacer distintas pruebas.

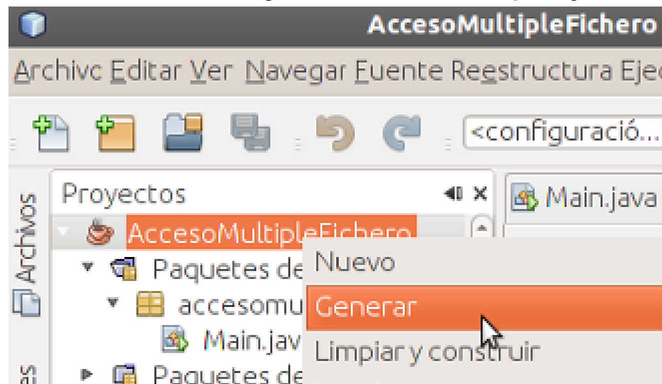
El código para tratar los argumentos recibidos por la línea de comandos, sería:

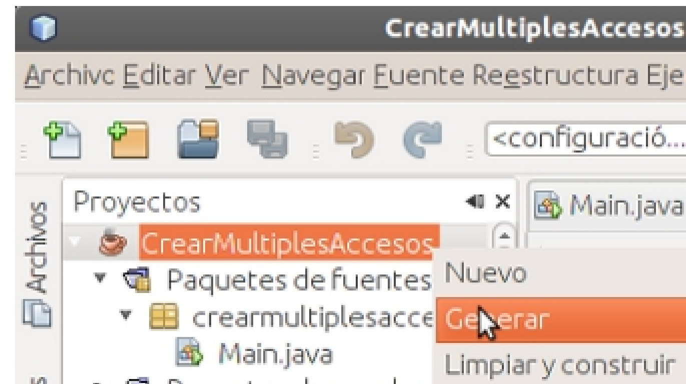
```
//Comprobamos si estamos recibiendo argumentos en la línea de comandos
if (args.length > 0){
    orden = Integer.parseInt(args[0]);
  }
//Identificamos el sistema operativo para poder acceder por su ruta al
//fichero de forma correcta.
String osName = System.getProperty("os.name");
if (osName.toUpperCase().contains("WIN")){ //Windows
  if (args.length > 1)
      nombreFichero = args[1].replace("\\", "\\\\");
      //Hemos recibido la ruta del fichero en la línea de comandos
  else{
      nombreFichero = "C:\\valor.txt";
      //Fichero que se utilizará por defecto
    }
}else{ //GNU/Linux
    if (args.length > 1)
      nombreFichero = args[1];
    //Hemos recibido la ruta del fichero en la línea de comandos
    else{
         nombreFichero = "/home/margye/valor.txt";
         //Fichero que se utilizará por defecto
  }
}
```

Probar la ejecución con varios procesos

Para probar nuestro ejemplo con los procesos que lo forman, haremos lo siguiente:

1. Generar los ficheros .jar de ambos proyectos.





2. Copiar los archivos .jar de ambos proyectos en la misma carpeta.
3. Lanzar la ejecución desde un terminal de comandos.

Lanzamos la ejecución desde línea de comandos, para que los ejecutables tomen como directorio de trabajo, el directorio en el que nos encontramos.

En Windows:

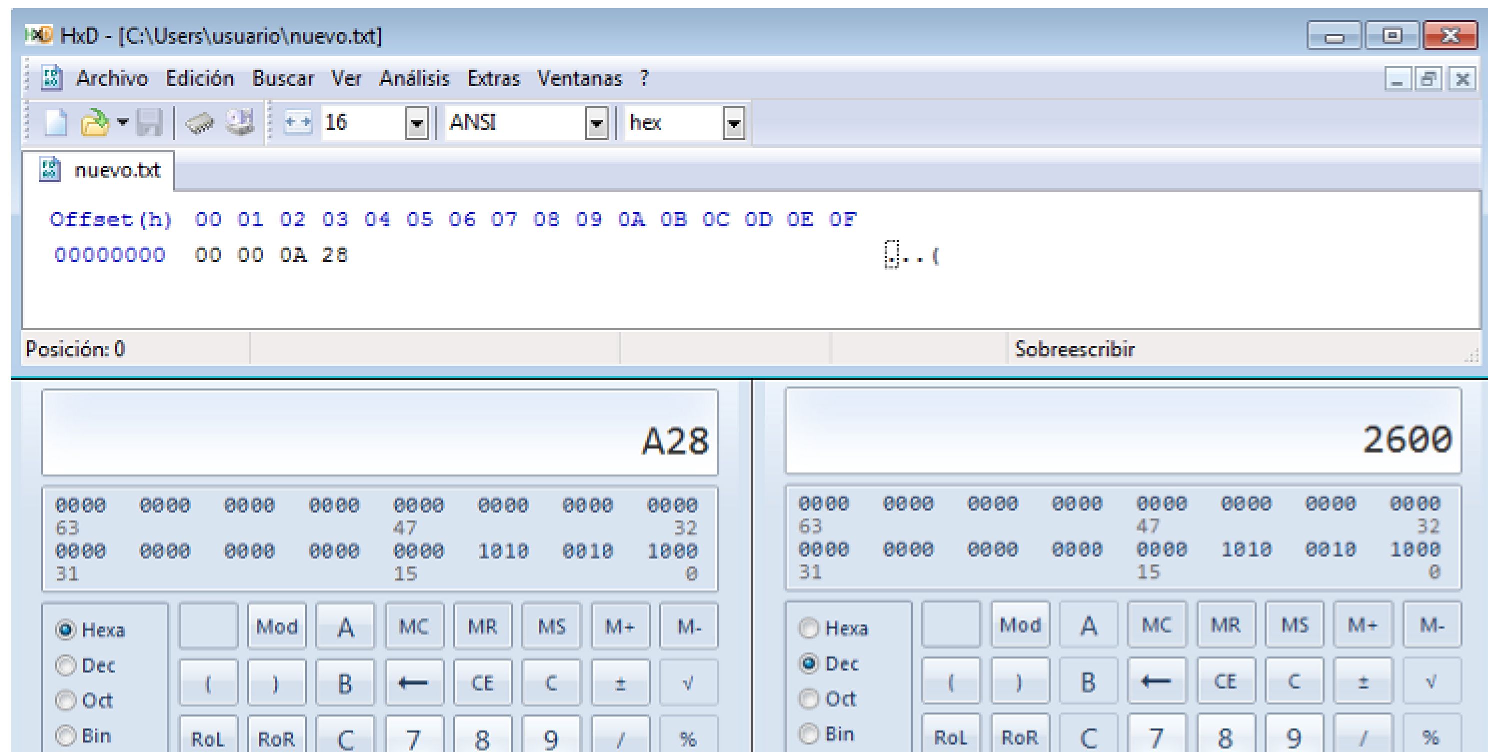
```
C:\Users\usuario\Proyectos_AccesosFicheroConSincro>java -jar
CrearMultiplesAccesos.jar c:\users\usuario\nuevo.txt
```

En GNU/Linux:

```
usuario@Pearl:~/AccesosFicherosConSincro$ java -jar
CrearMultiplesAccesos.jar nuevo.txt
```

Probar la ejecución con varios procesos

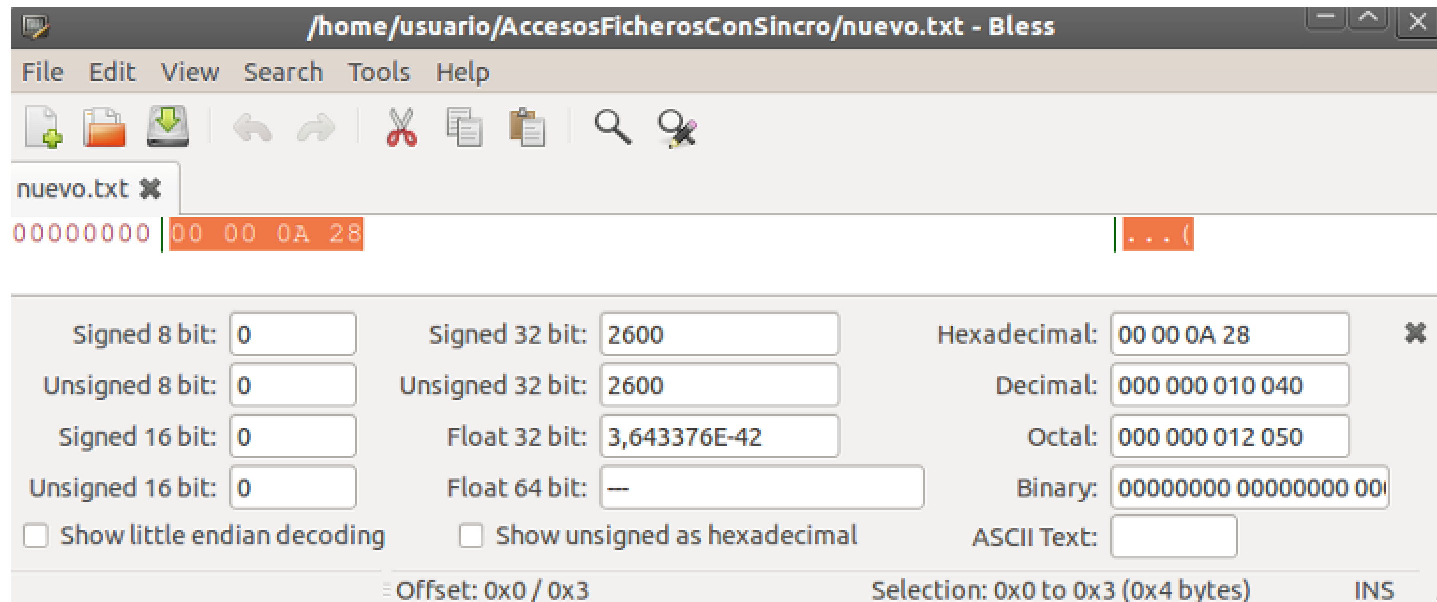
Revisamos el contenido del fichero que han estado utilizando los procesos.
En este ejemplo, en lugar de utilizar ficheros de texto y guardar el número como cadena ASCII; los procesos han guardado el valor en formato binario. Veremos el valor guardado en el fichero utilizando nuestros editores hexadecimales. HxD (Freeware) en sistemas Windows.
En Windows:



Hacemos uso de la calculadora de Windows para pasar el valor de hexadecimal a decimal.
Si hemos creado 26 procesos y cada uno incrementa 100 veces el valor del fichero comenzando por 0, al final debe ser 2600 ó 0A28 en hexadecimal.

Probar la ejecución con varios procesos

Revisamos el contenido del fichero que han estado utilizando los procesos.
En este ejemplo, en lugar de utilizar ficheros de texto y guardar el número como cadena ASCII; los procesos han guardado el valor en formato binario. Veremos el valor guardado en el fichero utilizando nuestros editores hexadecimales. Bless Hex Editor (GNU GPL v2) en sistemas GNU/Linux.
En GNU/Linux:



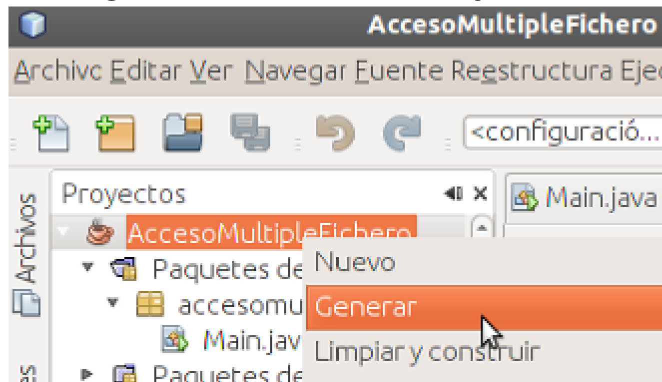
Si hemos creado 26 procesos y cada uno incrementa 100 veces el valor del fichero comenzando por 0, al final debe ser 2600 ó 0A28 en hexadecimal.

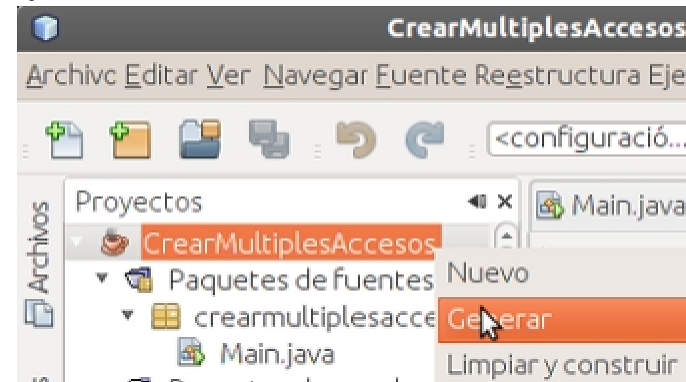
¿Cómo podemos ver lo que han hecho esos procesos?
Vamos a revisar el contenido de los ficheros javalog.txt.

Probar la ejecución con varios procesos escribiendo resultados en un fichero.

Para probar nuestro ejemplo con los procesos que lo forman, haremos lo siguiente:

1. Volvemos a generar los ficheros .jar de ambos proyectos.





2. Volvemos a copiar los archivos .jar de ambos proyectos en la misma carpeta.
3. Lanzamos la ejecución desde un terminal de comandos.

Lanzamos la ejecución desde línea de comandos, para que los ejecutables tomen como directorio de trabajo, el directorio en el que nos encontramos (se crearán los ficheros en el directorio actual).

En Windows:

```
C:\Users\usuario\Proyectos_AccesosFicheroConSincro>java -jar
CrearMultiplesAccesos.jar c:\users\usuario\nuevo.txt
C:\Users\usuario\AccessoMultiplesFichero>dir
 Directorio de C:\Users\usuario\AccessoMultiplesFichero
20/08/2011  12:42            4.900 AccesoMultipleFichero.jar
20/08/2011  10:24            2.529 CrearMultiplesAccesos.jar
20/08/2011  13:13            2.201 javalog.txt
20/08/2011  13:13                4 nuevo.txt
               4 archivos          9.634 bytes
               2 dirs   1.555.165.184 bytes libres
```

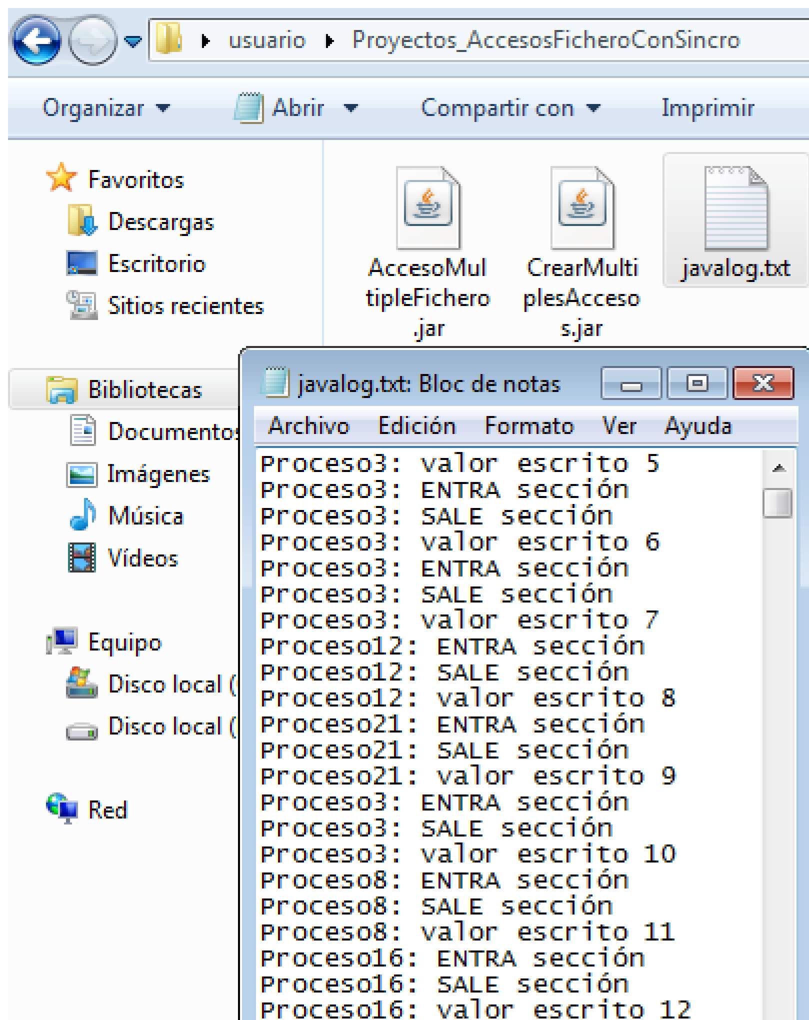
En GNU/Linux:

```
usuario@Pearl:~/AccesosFicherosConSincro$ ls
AccesoMultipleFichero.jar  CrearMultiplesAccesos.jar
javalog.txt  nuevo.txt
usuario@Pearl:~/AccesosFicherosConSincro$
```

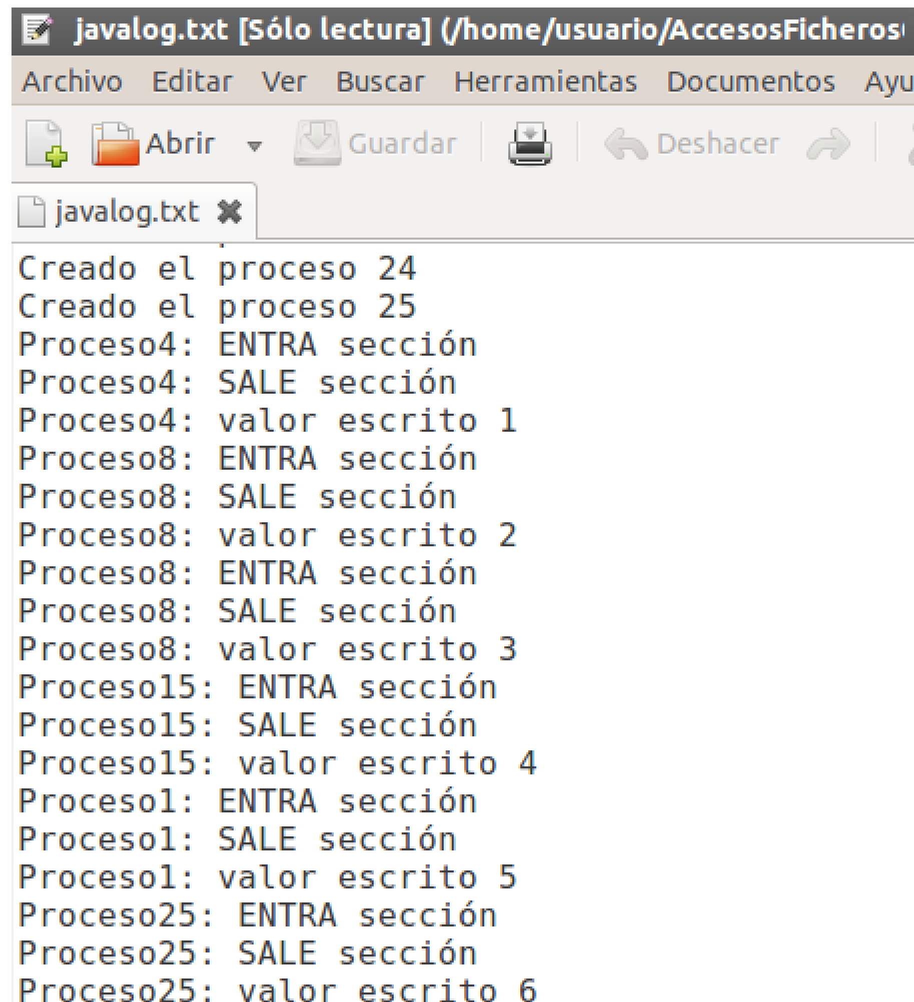
Ver lo que ha estado pasando

El valor final es el que esperábamos. Pero, ¿los procesos han estado respetando la sección crítica? Revisemos el contenido del fichero javalog.txt:

En Windows:



En GNU/Linux:



En todos los casos, no hay dos “ENTRA sección” consecutivos de procesos distintos.

Recomendaciones para probar el ejemplo

Si no has ido construyendo paso a paso el ejemplo, puedes probar el ejemplo que te suministramos implementado en la plataforma.

Recuerda:

1. Generar cada uno de los proyectos de forma independiente.

2. Copiar los archivos .jar resultantes de la generación en el mismo directorio.

3. Lanzar desde el terminal de comandos el ejecutable CrearMultiplesAccesos.jar.
   - El comando será: java -jar CrearMultiplesAccesos.jar rutaArchivo

4. Abrir los archivos rutaArchivo y javalog.txt que generan la ejecución de este ejemplo.
   Recuerda que en este caso, el valor se guarda en el archivo en formato binario (en lugar de formato texto). Por lo que para comprobar la corrección del valor final del fichero, tendrás que utilizar un editor hexadecimal.
   Podrás ver el contenido del fichero javalog.txt con cualquier editor de texto.

Credenciales

Imagen	Datos de licencia
Todas las capturas de pantalla de este presentación, tienen como datos de licencia: Autoría: Margarita I. Nieto Castillejo Licencia: Uso educativo-no comercial. Procedencia: Capturas de pantalla del IDE NetBeans 6.9.1; terminal de comandos, explorador de ficheros y Block de notas de Windows 7; consola de comandos, Nautilus y gedit de Ubuntu 10.10.; y, las herramientas HxD (Freeware), Bless Hex Editor (GNU GPL v2).	