LENGUAJES DE PROGRAMACIÓN

(Sesión 5)

3. PROGRAMACIÓN CONCURRENTE

3.1. La concurrencia en los lenguajes de programación

3.2. Soporte para la concurrencia en Java: Threads

Objetivo: Establecer el concepto de concurrencia y determinar la estructura general de un lenguaje concurrente.

INTRODUCCION

Actualmente observamos que el paradigma orientado a objetos, solo podemos ejecutar un equipo a la vez como máximo en cambio con la introducción de las hebras concurrentes(programación concurrente) o procesos es posible que cada objeto se ejecute simultáneamente, esperando mensajes y respondiendo adecuadamente. Como siempre la principal razón para la investigación de la programación concurrente es que nos ofrece una manera diferente de conceptualizar la solución de un problema, una segunda razón es la de aprovechar el paralelismo del hardware subyacente para lograr una aceleración significativa.

Para entender mejor este detalle un buen ejemplo de un programa concurrente es el navegador Web de modem. Un ejemplo de concurrencia en un navegador Web se produce cuando el navegador empieza a presentar una página aunque puede estar aun descargando varios archivos de gráficos o de imágenes. La página que estamos presentando es un recurso compartido que deben gestionar cooperativamente las diversas hebras involucradas en la descarga de todos los aspectos de una página. Las diversas hebras no pueden escribir todas en la pantalla simultáneamente, especialmente si la imagen o grafico descargado provoca el cambio de tamaño del espacio asignado a la visualización de la imagen, afectando así la distribución del texto. Mientras hacemos todo esto hay varios botones que siguen activos sobre los que podemos hacer click particularmente el boton stop como una suerte de conclusión se observa en este paper que las hebras operan para llevar a cabo una tarea como la del ejemplo anterior así mismo se vera que los procesos deben tener acceso exclusivo aun recurso compartido como por ejemplo la visualización para evitar interferir unas con otras.

CONCEPTOS:

Ø PROGRAMA SECUENCIAL: Es aquel que especifica la ejecución de una secuencia de instrucciones que comprenden el programa.

Ø PROCESO: Es un programa en ejecución, tiene su propio estado independiente del estado de cualquier otro programa incluidos los del sistema operativo. Va acompañado de recursos como archivos, memoria, etc.

Ø PROGRAMA CONCURRENTE: Es un programa diseñado para tener 2 o mas contextos de ejecución decimos que este tipo de programa es multihenbrado, porque tiene mas de un contexto de ejecución.

Ø PROGRAMA PARALELO: Es un programa concurrente en el que hay mas de un contexto de ejecución o hebra activo simultáneamente; desde un punto de vista semántica no hay diferencia entre un programa paralelo y concurrente.

Ø PROGRAMA DISTRIBUIDO: Es un sistema diseñado para ejecutarse simultáneamente en una red de procesadores autónomos, que no comparten la memoria principal, con cada programa en su procesador aparte.

En un sistema operativo de multiprocesos el mismo programa lo pueden ejecutar múltiples procesos cada uno de ellos dando como resultado su propio estado o contexto de ejecución separado de los demás procesos. Ejemplo:

En la edición de documentos o arhivos de un programa en cada instancia del editor se llama de manera separada y se ejecuta en su propia ventana estos es claramente diferente de un programa multienhebrado en el que alguno de los datos residen simultáneamente en cada contexto de ejecución:

FUNDAMENTOS DE LA PROGRAMACIÓN CONCURRENTE

Concurrencia: Es un termino genérico utilizado para indicar un programa único en el que puede haber mas de un contexto de ejecución activo simultáneamente.

Comunicación entre procesos

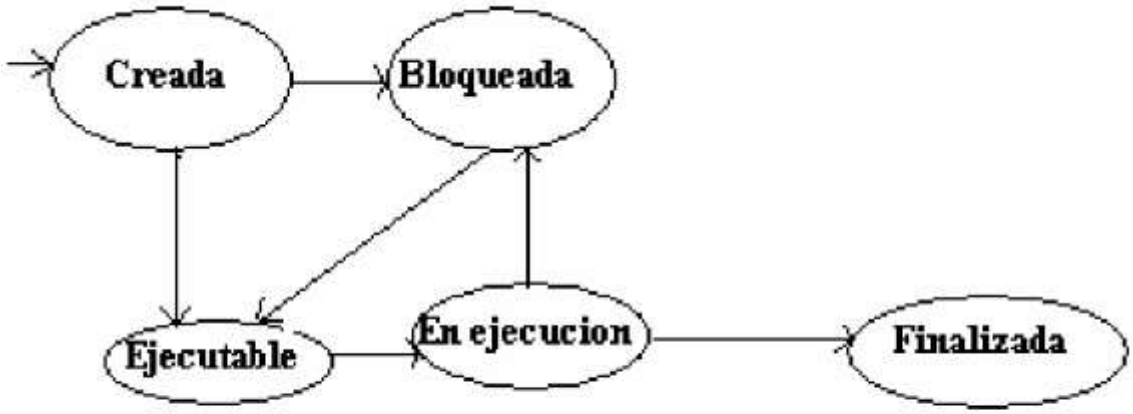
Introducción.-

Las aplicaciones informáticas pueden ser de muchos tipos desde procesadores de texto hasta procesadores de transacciones (Base de datos), simuladores de calculo intensivo, etc.

Las aplicaciones están formadas de uno o más programas. Los programas constan de código para la computadora donde se ejecutaran, el código es generalmente ejecutado en forma secuencial es así que normalmente un programa hilado o hebrado tiene el potencial de incrementar el rendimiento total de la aplicación en cuanto a productividad y tiempo de respuesta mediante ejecución de código asíncrono o paralelo.

Estados de una hebra (PROCESO):

1. Creada: La hebra se ha creado pero aun no esta lista para ejecutarse.
2. Ejecutable o lista: La hebra esta lista para ejecutarse pero espera a conseguir un procesador en que ejecutarse.
3. En ejecución: La hebra se esta ejecutando en un procesador.
4. Bloqueado o en espera: La hebra esta esperando tener acceso a una sección critica o ha dejado el procesador voluntariamente.
5. Finalizada: La hebra se ha parado y no se volverá a ejecutar.



Análisis de la comunicación entre procesos

Todos los programas concurrentes, implican interacción o comunicación entre hebras. Esto ocurre por las siguientes razones:

Ø Las hebras (incluso los procesos), compiten por un acceso exclusivo a los recursos compartidos, como los archivos físicos: archivos o datos.

Ø Las hebras se comunican para intercambiar datos.

En ambos casos es necesario que las hebras sincronicen su ejecución para evitar conflictos cuando adquieren los recursos, o para hacer contacto cuando intercambian datos. Una hebra

puede comunicarse con otras mediante:

Ø Variables compartidas no locales: es el mecanismo principal utilizado por JAVA y también puede utilizarlo ADA.

Ø Paso de mensajes: es el mecanismo utilizado por ADA.

Ø Parámetro: lo utiliza ADA junto con el pasote mensajes.

Las hebras cooperan unas con otras para resolver un problema.

Exclusión mutua:

El método más sencillo de comunicación entre los procesos de un programa concurrente es el uso común de unas variables de datos. Esta forma tan sencilla de comunicación puede llevar, no obstante, a errores en el programa ya que el acceso concurrente puede hacer que la acción de un proceso interfiera en las acciones de otro de una forma no adecuada. Aunque nos vamos a fijar en variables de datos, todo lo que sigue sería válido con cualquier otro recurso del sistema que sólo pueda ser utilizado por un proceso a la vez. Por ejemplo una variable x compartida entre dos procesos A y B que pueden incrementar o decrementar la variable dependiendo de un determinado suceso. Éste situación se plantea, por ejemplo, en un problema típico de la programación concurrente conocido como el Problema de los Jardines.

En este problema se supone que se desea controlar el número de visitantes a unos jardines. La entrada y la salida a los jardines se pueden realizar por dos puntos que disponen de puertas giratorias. Se desea poder conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un computador con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida. Asociamos el proceso P1 a un punto de entrada y el proceso P2 al otro punto de entrada. Ambos procesos se ejecutan de forma concurrente y utilizan una única variable x para llevar la cuenta del número de visitantes. El incremento o decremento de la variable se produce cada vez que un visitante entra o sale por una de las puertas. Así, la entrada de un visitante por una de las puertas hace que se ejecute la instrucción x:=x+1 mientras que la salida de un visitante hace que se ejecute la instrucción x:=x-1. Si ambas instrucciones se realizaran como una única instrucción hardware, no se plantearía ningún problema y el programa podría funcionar siempre correctamente. Esto es así por que en un sistema con un único procesador sólo se puede realizar una instrucción cada vez y en un sistema multiprocesador se arbitran mecanismos que impiden que varios procesadores accedan a la vez a una misma posición de memoria. El resultado sería que el incremento o decremento de la variable se produciría de forma secuencial pero sin interferencia de un proceso en la acción del otro. Sin embargo, sí se produce interferencia de un proceso en el otro si la actualización de la variable se realiza mediante la ejecución de otras instrucciones más sencillas,

como son las usuales de:

Ø copiar el valor de x en un registro del procesador.

Ø incrementar el valor del registro

Ø almacenar el resultado en la dirección donde se guarda x

Aunque el proceso P1 y el P2 se suponen ejecutados en distintos procesadores (lo que no tiene porque ser cierto en la realidad) ambos usan la misma posición de memoria para guardar el valor de x.

Semáforos:

Se definieron originalmente en 1968 por dijkstra, fue presentado como un nuevo tipo de variable. Dijkstra una solución al problema de la exclusión mutua con la introducción del concepto de semáforo binario. Está técnica permite resolver la mayoría de los problemas de sincronización entre procesos y forma parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes.

Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no. Un semáforo binario sólo puede tomar dos valores: 0 y 1. Si, para un semáforo binario, S = 1 entonces el recurso está disponible y la tarea lo puede utilizar; si S = 0 el recurso no está disponible y el proceso debe esperar.

Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso.

Sólo se permiten tres operaciones sobre un semáforo

Ø Inicializar

Ø Espera (wait)

Ø Señal (signal)

En algunos textos, se utilizan las notaciones P y V (DOWN y UP) para las operaciones de espera y señal respectivamente, ya que ésta fue la notación empleada originalmente por Dijkstra para referirse a las operaciones. Así pues, un semáforo binario se puede definir como un tipo de datos especial que sólo puede tomar los valores 0 y 1, con una cola de tareas asociada y con sólo tres operaciones para actuar sobre él.

Originalmente en el articulo de dijkstra, se utilizaron los nombres de P y V en lugar de down y up, pero ahora se utiliza los últimos que fueron presentados en ALGOL 60.

Interbloqueo e Injusticia

Se dice que una hebra esta en estado de ínter bloqueo, si esta esperando un evento que no se producirá nunca, el ínter bloqueo implica varias hebras, cada una de ellas a la espera de recursos existentes en otras. Un ínter bloqueo puede producirse, siempre que dos o más hebras

compiten por recursos; para que existan ínter bloqueos son necesarias 4 condiciones:

Ø Las hebras deben reclamar derechos exclusivos a los recursos.

Ø Las hebras deben contener algún recurso mientras esperan otros; es decir, adquieren los recursos poco a poco, en vez de todos a la vez.

Ø No se pueden sacar recursos de hebras que están ala espera (no hay derecho preferente).

Ø Existe una cadena circular de hebras en la que cada hebra contiene uno o mas recursos necesarios para la siguiente hebra de la cadena

Sincronización

El uso de semáforos hace que se pueda programar fácilmente la sincronización entre dos tareas. En este caso las operaciones espera y señal no se utilizan dentro de un mismo proceso sino que se dan en dos procesos separados; el que ejecuta la operación de espera queda bloqueado hasta que el otro proceso ejecuta la operación de señal.

A veces se emplea la palabra señal para denominar un semáforo que se usa para sincronizar procesos. En este caso una señal tiene dos operaciones: espera y señal que utilizan para sincronizarse dos procesos distintos.

Monitores

Un monitor es un conjunto de procedimientos que proporciona el acceso con exclusión mutua a un recurso o conjunto de recursos compartidos por un grupo de procesos. Los procedimientos van encapsulados dentro de un módulo que tiene la propiedad especial de que sólo un proceso puede estar activo cada vez para ejecutar un procedimiento del monitor.

El monitor se puede ver como una valla alrededor del recurso (o recursos), de modo que los procesos que quieran utilizarlo deben entrar dentro de la valla, pero en la forma que impone el monitor. Muchos procesos pueden querer entrar en distintos instantes de tiempo, pero sólo se permite que entre un proceso cada vez, debiéndose esperar a que salga el que está dentro antes de que otro pueda entrar.

La ventaja para la exclusión mutua que presenta un monitor frente a los semáforos u otro mecanismo es que ésta está implícita: la única acción que debe realizar el programador del proceso que usa un recurso es invocar una entrada del monitor. Si el monitor se ha codificado correctamente no puede ser utilizado incorrectamente por un programa de aplicación que desee usar el recurso. Cosa que no ocurre con los semáforos, donde el programador debe proporcionar la correcta secuencia de operaciones espera y señal para no bloquear al sistema.

Los monitores no proporcionan por si mismos un mecanismo para la sincronización de tareas y por ello su construcción debe completarse permitiendo, por ejemplo, que se puedan usar señales para sincronizar los procesos. Así, para impedir que se pueda producir un bloqueo, un proceso

que gane el acceso a un procedimiento del monitor y necesite esperar a una señal, se suspende y se coloca fuera del monitor para permitir que entre otro proceso.

Una variable que se utilice como mecanismo de sincronización en un monitor se conoce como variable de condición. A cada causa diferente por la que un proceso deba esperar se asocia una variable de condición. Sobre ellas sólo se puede actuar con dos procedimientos: espera y señal. En este caso, cuando un proceso ejecuta una operación de espera se suspende y se coloca en una cola asociada a dicha variable de condición. La diferencia con el semáforo radica en que ahora la ejecución de esta operación siempre suspende el proceso que la emite. La suspensión del proceso hace que se libere la posesión del monitor, lo que permite que entre otro proceso. Cuando un proceso ejecuta una operación de señal se libera un proceso suspendido en lacola de la variable de condición utilizada. Si no hay ningún proceso suspendido en la cola de lavariable de condición invocada la operación señal no tiene ningún efecto.

Mensajes

Los mensajes proporcionan una solución al problema de la concurrencia de procesos que integra la sincronización y la comunicación entre ellos y resulta adecuado tanto para sistemas centralizados como distribuidos. Esto hace que se incluyan en prácticamente todos lo ssistemas operativos modernos y que en muchos de ellos se utilicen como base para todas las comunicaciones del sistema, tanto dentro del computador como en la comunicación entre computadores.

La comunicación mediante mensajes necesita siempre de un proceso emisor y de uno receptor así como de información que intercambiarse. Por ello, las operaciones básicas para comunicación mediante mensajes que proporciona todo sistema operativo son: enviar (mensaje) y recibir (mensaje). Las acciones de transmisión de información y de sincronización se ven como actividades inseparables.

La comunicación por mensajes requiere que se establezca un enlace entre el receptor y el emisor, la forma del cual puede variar grandemente de sistema a sistema. Aspectos importantes a tener en cuenta en los enlaces son: como y cuantos enlaces se pueden establecer entre los procesos, la capacidad de mensajes del enlace y tipo de los mensajes.

Su implementación varía dependiendo de tres aspectos:

Ø El modo de nombrar los procesos.

Ø El modelo de sincronización.

Ø Almacenamiento y estructura del mensaje.

ANALISIS DE CONCEPTOSMEDIANTE LOS EJEMPLOS CLASICOS DE LOS PROCESOS CONCURRENTES

El problema de la cena de los filósofos

En 1965 dijsktra planteo y resolvió un problema de sincronización llamado el problema de la cena de los filósofos, desde entonces todas las personas que idean cierta primitiva de sincronización, intentan demostrar lo maravilloso de la nueva primitiva al resolver este problema.

Se enuncia de la siguiente manera:

"5 filósofos se sientan en la mesa, cada uno tiene un plato de spaghetti, el spaghetti es tan escurridizo que un filósofo necesita dos tenedores para comerlo, entre cada dos platos hay un tenedor. La vida de un filosofo, consta de periodos alternados de comer y pensar, cuando un filosofo tiene hambre, intenta obtener un tenedor para su mano izquierda y otro para su mano derecha, alzando uno a la vez y en cualquier orden, si logra obtener los dos tenedores, come un rato y después deja los tenedores y continua pensando, la pregunta clave es: ¿puede usted escribir un programa, para cada filosofo que lleve a cabo lo que se supone debería y que nunca se detenga?, la solución obvia para este problema es que el procedimiento espere hasta que el tenedor especificado este disponible y se toman . Por desgracia esta solución es incorrecta.

Supongamos que los 5 filósofos toman sus tenedores izquierdos en forma simultánea. Ninguno de ellos podría tomar su tenedor derecho con lo que ocurriría un bloqu**eo"**.

Una situación como esta, en la que todos los programas se mantendrían por tiempo indefinido, pero sin hacer progresos se llama inanición

Decimos que una hebra esta aplazada indefinidamente si se retraza esperando un evento que puede no ocurrir nunca. La asignación de los recursos sobre una base de primero en entrar – primero en salir es una solución sencilla que elimina el desplazamiento indefinido.

Análogo al desplazamiento indefinido es el concepto de injusticia en este caso no se realiza ningún intento para asegurarse de que las hebras que tienen el mismo status hacen el mismo progreso en la adquisición de recursos, o sea, no toda acciones son igualmente probable.

Problema de los lectores y escritores:

Este problema modela el acceso a una base de datos. Imaginemos una enorme base de datos, como por ejemplo un sistema de reservaciones en una línea aérea con muchos procesos en competencia que intentan leer y escribir en ella.

Se puede aceptar que varios procesos lean la base de datos al mismo tiempo, pero si uno de los procesos esta escribiendo la base de datos, ninguno de los demás procesos debería tener acceso a esta, ni siquiera los lectores .La pregunta es: ¿Cómo programaría usted los lectores y escritores?.

Una posible solución es que el primer lector que obtiene el acceso, lleva a cabo un down

En el semáforo dv. Los siguientes lectores solo incrementan un contador. Al salir los

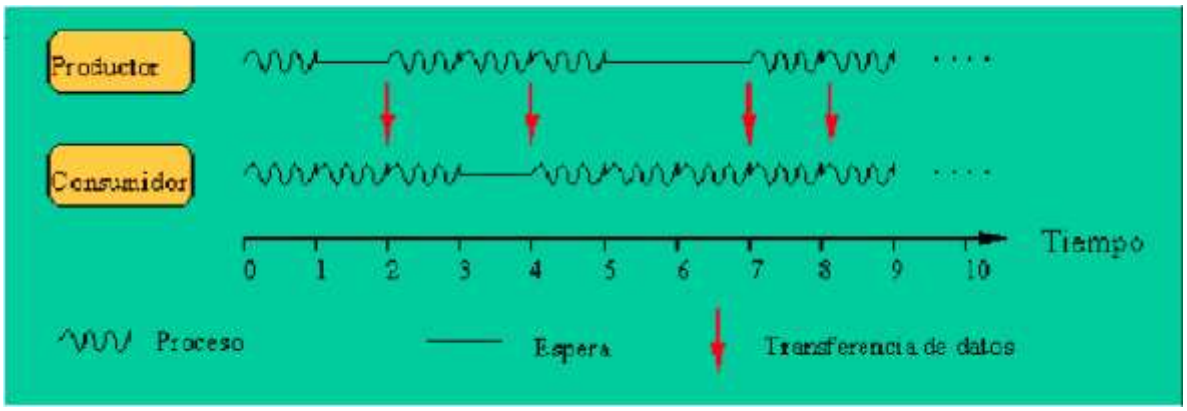
lectores estos decrementan el contador y el ultimo en salir hace un up en el semáforo, lo cual permite entrar a un escritor bloqueado si es que existe. Una hipótesis implícita en esta solución es que los lectores tienen prioridad entre los escritores, si surge un escritor mientras varios lectores se encuentran en la base de datos, el escritor debe esperar.
Pero si aparecen nuevos lectores, de forma que exista al menos un lector en la base de datos, el escritor deberá esperar hasta que no haya más lectores.
Este problema genero muchos puntos de vista tanto así que courtois etal también presento una solución que da prioridad a los escritores.

El problema del barbero dormilón:
Otro de los problemas clásicos de este paradigma ocurre en una peluquería, la peluquería tiene un barbero, una silla de peluquero y n sillas para que se sienten los clientes en espera, si no hay clientes presentes el barbero se sienta y se duerme. Cuando llega un cliente, este debe despertar al barbero dormilón. Si llegan mas clientes mientras el barbero corta el cabello de un cliente, ellos se sientan (si hay sillas desocupadas), o en todo caso, salen de la peluquería. El problema consiste en programar al barbero y los clientes sin entrar en condiciones de competencia. Una solución seria: cuando el barbero abre su negocio por la mañana ejecuta el procedimiento
barber, lo que establece un bloqueo en el semáforo "**customers"**, hasta que alguien llega,
después se va a dormir. Cuando llega el primer cliente el ejecuta "customer**"**. Si otro cliente llega poco tiempo después, el segundo no podrá hacer nada. Luego verifica entonces si el número de clientes que esperan es menor que el número de sillas, si esto no ocurre, sale sin su corte de pelo. Si existe una silla disponible, el cliente incrementa una variable contadora. Luego realiza un up en el semáforo "customer**"** con lo que despierta al barbero. En este momento tanto el cliente como el barbero están despiertos, luego cuando le toca su turno al cliente le cortan el pelo.

El problema del productor-consumidor. Estudiaremos problemas en la comunicación, supongamos dos tipos de procesos: Productores y Consumidores.

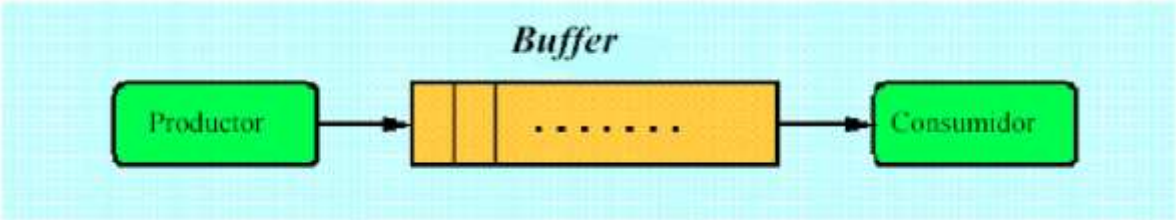
(Produce), estos datos deben ser enviados a los consumidores.

Consumidores: Procesos que reciben los elementos de datos creados por los productores, y que actúan en consecuencia mediante un procedimiento interno (Consume). Ejemplos: Impresoras, teclados, etc.

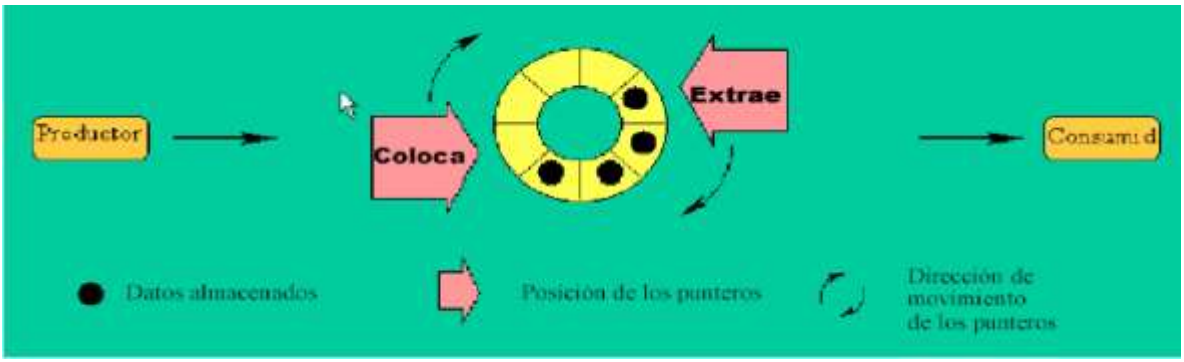


Problema: El productor envía un dato cada vez, y el consumidor consume un dato cada vez. Si uno de los dos procesos no está listo, el otro debe esperar.

Solución: Es necesario introducir un buffer en el proceso de transmisión de datos

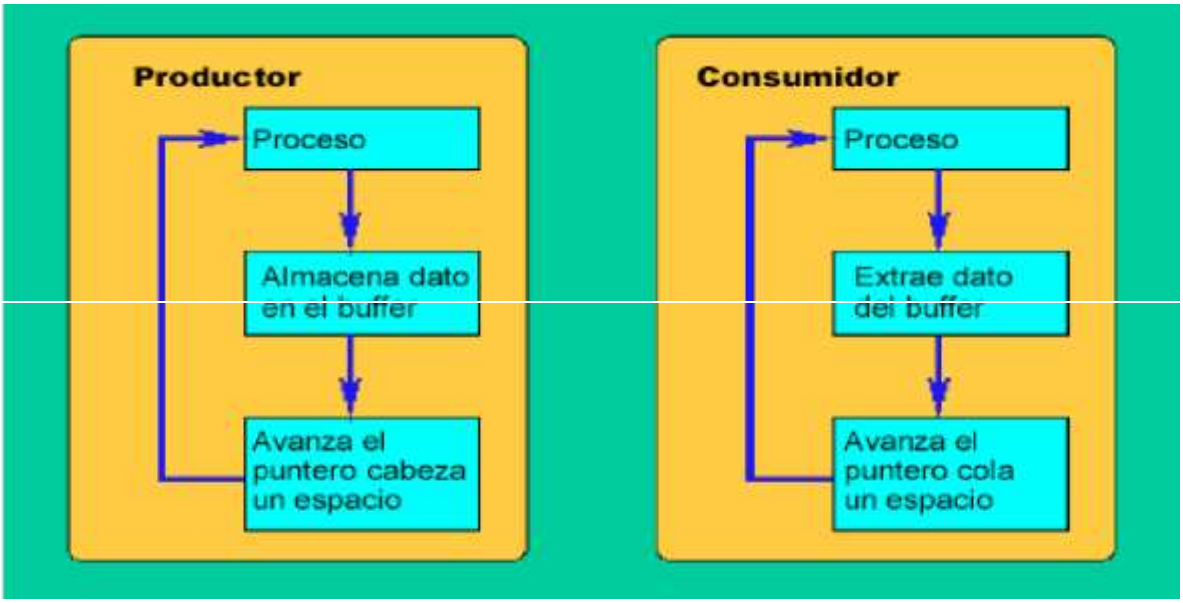


El buffer puede ser infinito. No obstante esto no es realista



Alternativa: Buffer acotado en cola circular

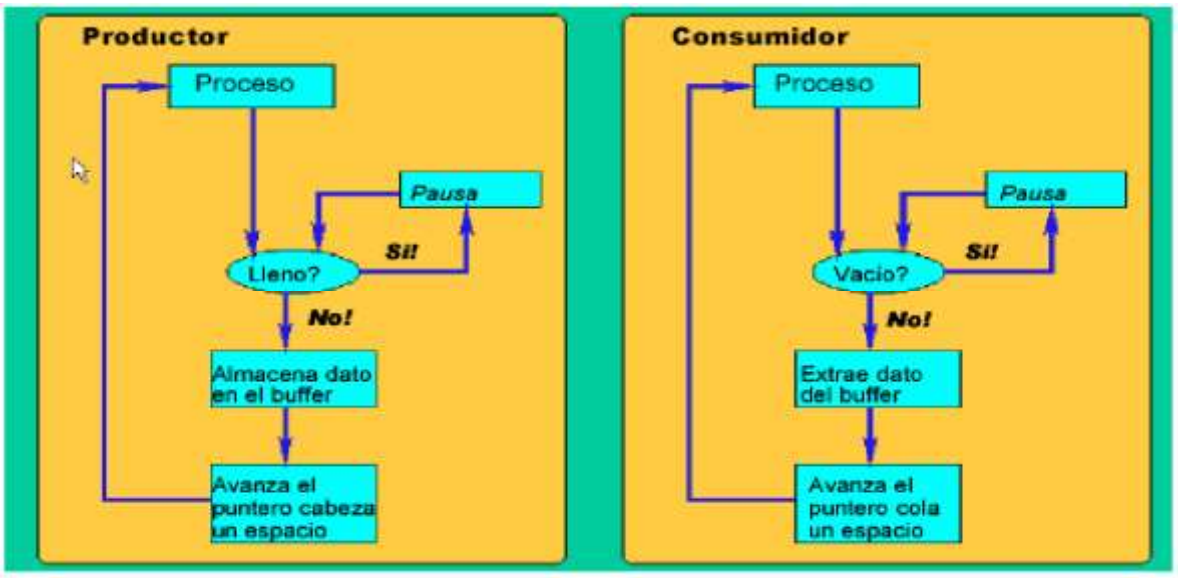
Algoritmo de funcionamiento del buffer acotado



Ø El productor puede enviar un dato a un buffer lleno

Ø El consumidor puede extraer un dato de un buffer vacío

ES NECESARIO PREVENIR ESTAS SITUACIONES ANÓMALAS: Algoritmo de funcionamiento del buffer acotado



¿Cómo saber si el buffer está vacío o lleno?

Ø Una condición será un semáforo inicializado a un cierto valor.

Ø Necesitamos dos condiciones: lleno y vacío.

Ø Para añadir un dato al buffer, es necesario comprobar (wait) la condición lleno. SI se efectúa la operación con éxito se realiza un signal sobre la condición vacío.

Ø Para eliminar un dato del buffer, es necesario comprobar (wait) la condición vacío. SI se efectúa la operación con éxito se realiza un signal sobre la condición lleno.

CARACTERISTICASDE LOS PROCESOS CONCURRENTES:

Interacción entre procesos: Los programas concurrentes implican interacción, entre los distintos procesos que los componen:

Ø Los procesos que compartan recursos y compiten por el acceso a los mismos.

Ø Los procesos que se comunican entre sí para intercambiar datos.

En ambas situaciones se necesitan que los procesos sincronicen su ejecución, para evitar

conflictos o establecer contacto para el intercambio de datos. Esto se logra mediante variables compartidas o bien mediante el paso de mensajes.

Indeterminismo: las acciones que se especifican en un programa secuencial, tienen un orden total, pero en un programa concurrente el orden es parcial, ya que existe una incertidumbre sobre el orden exacto de concurrencia de ciertos sucesos. De esta forma si se ejecuta un programa concurrente varias veces, puede producir resultados diferentes partiendo de los mismos datos.

Gestión de recursos: los recursos compartidos necesitan una gestión especial. Un proceso que desea utilizar un recurso compartido debe solicitar dicho recurso, esperar a adquirirlo, utilizarlo y después liberarlo. Si el proceso solicita el recurso, pero no puede adquirirlo, en ese momento, es suspendido hasta que el recurso esta disponible.

La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retrazo indefinido o de bloqueo indefinido.

Comunicación: la comunicación entre procesos puede ser síncrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor lo recoja, ya que los deja en un buffer de comunicación temporal.

Violación de la exclusión mutua: en ocasiones ciertas acciones que se realizan en un programa concurrente, no proporcionan los resultados deseados.

Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una región critica, es decir, es una parte de un programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder.

Bloqueo mutuo: un proceso se encuentra en estado de bloqueo mutuo si esta esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y en la gestión de

recursos. Se basan en las siguientes condiciones:

Ø Los procesos necesitan acceso exclusivo a los recursos.
Ø Los procesos necesitan mantener ciertos recursos exclusivos y otros en espera.
Ø Los recursos no se pueden obtener de los procesos que están a la espera.

PROGRAMACION EN JAVA

Hebras de java:

Una hebra puede estar en uno de estos cinco estados: creada, ejecutable, en ejecución, bloqueada o finalizada. Una hebra puede hacer transiciones de un estado a otro, ignorando en buena parte la transición del estado ejecutable al de la ejecución debido a que esto lo manipula el equipo virtual de java subyacente. En java como ocurre con todos los demás, una hebra es una clase por tanto la manera más sencilla de crear una hebra es crear una clase que herede de la clase thread:

```
public class mytrhead extends thread
{
public mythread()
{
……..
..……
}}
```

Y hacer una operación new para crear una instancia de una thread: Thread thread = new mythread ();

Para hacer que una hebra creada sea ejecutable, llamamos sencillamente a su método start:

```
thread.start ( );
```

Después de algún coste adicional, la hebra recientemente ejecutable transfiere el control a su método run cada clase que amplia la clase thread debe proporcionar su propio método thread, pero no facilita un método start, basándose en el método start facilitado por la clase thread, normalmente el método run de una hebra contiene un bucle, ya que la salida de el método run finaliza la hebra. Por ejemplo en una animación grafica el método run movería repetidamente los objetos gráficos, repintaría la pantalla y después se dormiría (para ralentizar la animación), por tanto una animación grafica normal podría ser:

```
public void run{
while ( true ){
```

```
movedObjects( );
repaint( );
try { thread:sleep(50);
} catch (InterruptedException exc){}
}
}
```

Observemos que el método sleep, lanza potencialmente un InterruptedExeption, que debe captarse en una instrucción try- catch .

La llamada al método sleep mueve la hebra del estado en ejecución a un estado bloqueado, donde espera una interrupción del temporalizador de intervalo. La acción de dormir se realiza con frecuencia en las aplicaciones visuales, para evitar que la visualización se produzca demasiado rápido.

Un modo de conseguir un estado de finalizada, es hacer que la hebra salga de su método run, lo que después de un retrazo para la limpieza finalizaría la hebra, también se puede finalizar una hebra, llamando su método stock. Sin embargo, por razones complicadas, esto resulto ser problemático y en Java se desprecio el método stop de la clase thread. Una manera más sencilla de conseguir el mismo objetivo es considerar un booleano que el método run de la hebra utiliza para continuar el bucle por ejemplo:

```
Public void run( ){ While (continue ){
moveObjects( );
repaint( );
try{ thread.sleep(50);
}catch (InterruptedExecption exc){}
}}
```

Para detener la hebra anterior, otra hebra llama a un método para establecer el valor de la variable de instancia continue en false. Esta variable no se considera compartida y podemos ignorar cualquier posible conexión de carrera de manera segura ya que como muchos provocara una interaccion extra del bucle.

A veces no es conveniente hacer sus clases de la clase thread , por ejemplo, podemos querer que nuestra clase applet sea una hebra separada. En estos casos, una clase solo tiene que implementar la interfaz runnabl; es decir implementar un método run, el esquema es de este tipo de clase es :

```
public class myclass extends someClass implements Runnable{
```

…….

Public void run( ){…..}

}

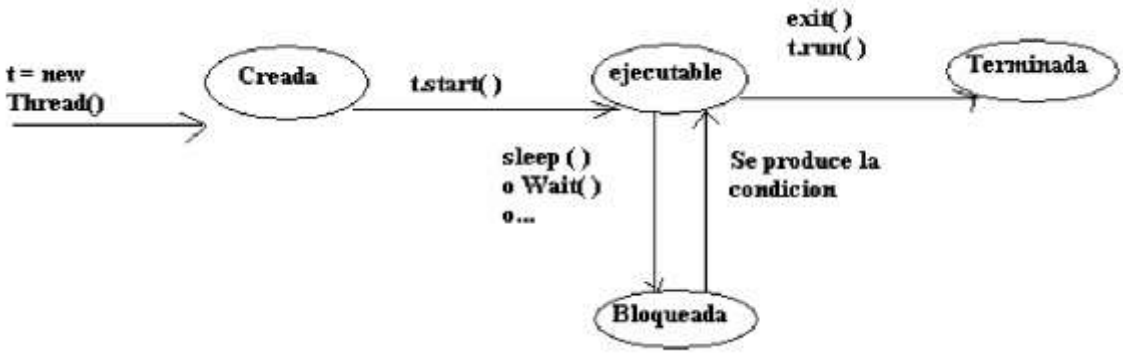
hacemos una hebra con una instancia de myclass utilizando:

myclass obj = new myclass;

thread thread = new thread (obj);

thread.start( );

Aquí vemos de nuevo una instancia en la que la utilización de interfaces de Java obvia la necesidad de herencia múltiple. La figura muestra los estados de una hebra de Java y las transiciones entre ellos:



SINCRONIZACIÓNEN JAVA:

Básicamente, Java implementa el concepto de monitor de una manera bastante rigurosa asociando un bloqueo con cada objeto. Para implementar una variable compartida, creamos una clase de variable compartida e indicamos cada método (excepto el constructor) como sincronizado:

Public class sharedVariable…{

Public sharedVariable(…){….}

Public synchronized…method (…){…} Public synchronized…method (…){…}

…}

para compartir una variable tenemos que crear una instancia de la clase y hacerla accesible para las hebras separadas . una manera de llevar estoa cabo seria pasar al objeto compartido, como parámetro del constructor de cada hebra. En el caso de una variable compartida de productorconsumidor, esto podría quedar así:

Shared Variable shared= new SharedVariable ( ); Thread producer = new Producer (shared);

Thread consumer = new (shared);

Donde damos por supuesto que tanto producerAndConsumer amplían switch.

PROGRAMACIÓN PARALELA

La programación paralela es una técnica de programación basada en la ejecución simultánea, bien sea en un mismo ordenador (con uno o varios procesadores) o en un cluster de ordenadores, en cuyo caso se denomina computación distribuida. Al contrario que en la programación concurrente, esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de las tareas.

Los sistemas multiprocesador o multicomputador consiguen un aumento del rendimiento si se utilizan estas técnicas. En los sistemas monoprocesador el beneficio en rendimiento no es tan evidente, ya que la CPU es compartida por múltiples procesos en el tiempo, lo que se denomina multiplexación.

El mayor problema de la computación paralela radica en la complejidad de sincronizar unas tareas con otras, ya sea mediante secciones críticas, semáforos o paso de mensajes, para garantizar la exclusión mutua en las zonas del código en las que sea necesario.

Es el uso de varios procesadores trabajando juntos para resolver una tarea común, cada procesador trabaja una porción del problema pudiendo los procesos intercambiar datos a través de la memoria o por una red de interconexión.

Un programa paralelo es un programa concurrente en el que hay más de un contexto de ejecución, o hebra, activo simultáneamente. Cabe recalcar que desde un punto de vista semántica no hay diferencia entre un programa concurrente y uno paralelo[3].

Necesidad de la programación paralela

Ø Resolver problemas que no caben en una CPU.

Ø Resolver problemas que no se resuelven en un tiempo razonable.

Ø Se puede ejecutar :

O Problemas mayores.

O Problemas mas rápidamente.

Ø El rendimiento de los computadores secuenciales esta comenzando a saturarse, una posible solución seria usar varios procesadores, sistemas paralelos, con la tecnología VLSI(Very Large Scale Integration), el costo de los procesadores es menor.

Ø Decrementa la complejidad de un algoritmo al usar varios procesadores.

Aspectos en la programación paralela

Ø Diseño de computadores paralelos teniendo en cuenta la escalabilidad y comunicaciones.

Ø Diseño de algoritmos eficientes, no hay ganancia si los algoritmos no se diseñan adecuadamente.

Ø Métodos para evaluar los algoritmos paralelos: ¿Cuán rápido se puede resolver un problema usando una máquina paralela?,¿Con que eficiencia se usan esos procesadores?.

Ø En lenguajes para computadores paralelos deben ser flexibles para permitir una implementación eficiente y fácil de programar.

Ø Los programas paralelos deben ser portables y los compiladores paralelizantes.

Modelos de computadoras

Ø Modelo SISD

1. Simple instrucción, simple Data.
2. Consiste en un procesador que recibe un cierto flujo de instrucciones y de datos.
3. Un algoritmo para esta clase dice ser secuencial o serial.

Ø ModeloMISD

4. Multiple Instruction, simpledata.
5. Existencia de N procesadores con un simple flujo de datos ejecutando instrucciones diferentes en cada procesador.

Ø Modelo SIMD

6. Simple Instruction, múltiple Data.
7. Existen JN procesadores ejecutando la misma instrucción sobre los mismos o diferentes datos.
8. Existencia de datos compartidos.

Ø ModeloMIMD

9. Multiple Instruction, multiple data.
10. Existen N procesadores ejecutando instrucciones diferentes sobre los mismos o diferentes datos.

11. Existencia de datos compartidos.

CONCLUSIONES:

Ø Un programa paralelo es un programa concurrente en el que hay más de un contexto de ejecución, o hebra, activo simultáneamente.

Ø Las aplicaciones de la concurrencia se centra principalmente en los sistemas operativos.

Ø Se ha propuesto varias primitivas de la comunicación entre procesos(Semáforos, monitores, etc) pero todas son equivalentes en el sentido de que cada una se puede utilizar para implementar a las demás.

Ø La programación paralela ayuda a disminuir la complejidad computacional de los algoritmos.

Ø La programación paralela nos ayuda a resolver problemas más complejos.

ANEXOS COMPLEMENTARIO

Aplicaciones de las hebras

Algunos programas presentan una estructura que puede hacerles especialmente adecuados para entornos multihilo. Normalmente estos casos involucran operaciones que pueden ser solapadas. La utilización de múltiples hilos, puede conseguir un grado de paralelismo que incremente el rendimiento de un programa, e incluso hacer más fácil la escritura de su código.

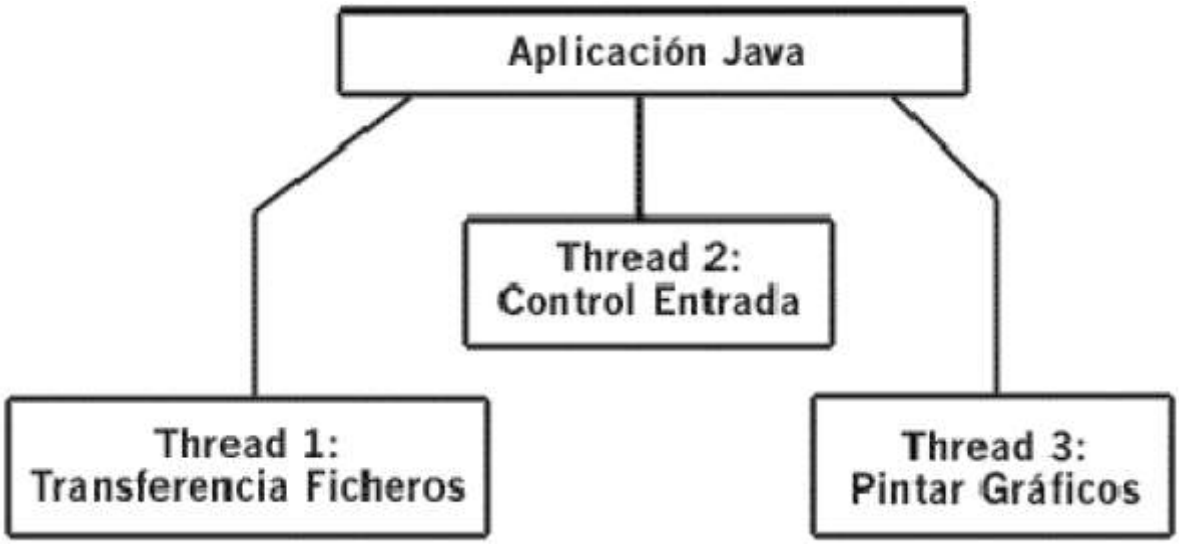
Algunos ejemplos son:

Ø Utilización de los hilos para expresar algoritmos inherentemente paralelos. Se pueden obtener aumentos de rendimiento debido al hecho de que los hilos funcionan muy bien en sistemas multiprocesadores. Los hilos permiten expresar el paralelismo de alto nivel a través de un lenguaje de programación.

Ø Utilización de los hilos para solapar operaciones de E/S lentas con otras operaciones en un programa. Esto permite obtener un aumento del rendimiento, permitiendo bloquear a un simple hilo que espera a que se complete una operación de E/S, mientras otros hilos del proceso continúan su ejecución, evitando el bloqueo entero del proceso.

Ø Utilización de los hilos para solapar llamadas RPC salientes. Se obtiene una ganancia en el rendimiento permitiendo que un cliente pueda acceder a varios servidores al mismo tiempo en lugar de acceder a un servidor cada vez. Esto puede ser interesante para servidores especializados en los que los clientes pueden dividir una llamada RPC compleja en varias llamadas RPC concurrentes y más simples.

Ø Utilización de los hilos en interfaces de usuario. Se pueden obtener aumentos de rendimiento empleando un hilo para interactuar con un usuario, mientras se pasan las peticiones a otros hilos para su ejecución.
Ø Utilización de los hilos en servidores. Los servidores pueden utilizar las ventajas del multihilo, creando un hilo gestor diferente para cada petición entrante de un cliente.
Ø Utilización de los hilos en procesos pipeline: Se puede implementar cada etapa de una tubería o pipeline mediante un hilo separado dentro del mismo proceso.
Ø Utilización de los hilos en el diseño de un kernel multihilo de sistema operativo distribuido que distribuya diferentes tareas entre los hilos.
Ø Utilización de los hilos para explotar la potencia de los multiprocesadores de memori a compartida (sistemas fuertemente acoplados).
Ø Utilización de los hilos como soporte de aplicaciones de tiempo real acelerando los tiempos de respuesta para los eventos asíncronos a través de la gestión de señales.
Algunas de estas técnicas pueden ser implementadas usando múltiples procesos. Los hilos son sin embargo más interesantes como solución porque:
Ø Los procesos tienen un alto coste de creación.
Ø Los procesos requieren más memoria.
Ø Los procesos tienen un alto coste de sincronización.
Ø Compartir memoria entre procesos es más complicado, aunque compartir memoria entre hilos tiene sus problemas.
Los hilos se crearon para permitir combinar el paralelismo con la ejecución secuencial y el bloqueo de llamadas al sistema.
Las llamadas al sistema con bloqueo facilitan la programación, y el paralelismo obtenido mejora el rendimiento.
Hilos y Multihilo
Considerando el entorno multithread (multihilo), cada thread (hilo, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. A veces se les llama procesos ligeros o contextos de ejecución. Típicamente, cada hilo controla un único aspecto dentro de un programa, como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todos los hilos comparten los mismos recursos, al contrario que los procesos, en donde cada uno tiene su propia copia de código y datos (separados unos de otros). Gráficamente, los hilos (threads) se parecen en su funcionamiento a lo que muestra la figura siguiente:



Hay que distinguir multihilo (multithread) de multiproceso. El multiproceso se refiere a dos programas que se ejecutan "aparentemente" a la vez, bajo el control del Sistema Operativo. Los programas no necesitan tener relación unos con otros, simplemente el hecho de que el usuario desee que se ejecuten a la vez.

Multihilo se refiere a que dos o más tareas se ejecutan "aparentemente" a la vez, dentro de un mismo programa.

Se usa "aparentemente" en ambos casos, porque normalmente las plataformas tienen una sola CPU, con lo cual, los procesos se ejecutan en realidad "concurrentemente", sino que comparten la CPU. En plataformas con varias CPU, sí es posible que los procesos se ejecuten realmente a la vez.

Tanto en el multiproceso como en el multihilo (multitarea), el Sistema Operativo se encarga de que se genere la ilusión de que todo se ejecuta a la vez. Sin embargo, la multitarea puede producir programas que realicen más trabajo en la misma cantidad de tiempo que el multiproceso, debido a que la CPU está compartida entre tareas de un mismo proceso. Además, como el multiproceso está implementado a nivel de sistema operativo, el programador no puede intervenir en el planteamiento de su ejecución; mientras que en el caso del multihilo, como el programa debe ser diseñado expresamente para que pueda soportar esta característica, es imprescindible que el autor tenga que planificar adecuadamente la ejecución de cada hilo, o tarea.

Actualmente hay diferencias en la especificación del intérprete de Java, porque el intérprete de Windows '95 conmuta los hilos de igual prioridad mediante un algoritmo circular (round-robin), mientras que el de Solaris 2.X deja que un hilo ocupe la CPU indefinidamente, lo que implica la inanición de los demás.

Programas de flujo único

Un programa de flujo único o mono-hilvanado (single-thread) utiliza un único flujo de control (thread) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchos de los applets y aplicaciones son de flujo único. Por ejemplo, en la archiconocida aplicación estándar de saludo:

```
public class HolaMundo {
static public void main( String args[] ) {
System.out.println( "Hola Mundo!" );
}
}
```

Aquí, cuando se llama a main(), la aplicación imprime el mensaje y termina. Esto ocurre dentro de un único hilo de ejecución (thread).

Debido a que la mayor parte de los entornos operativos no solían ofrecer un soporte razonable para múltiples hilos de control, los lenguajes de programación tradicionales, tales como C++, no incorporaron mecanismos para describir de manera elegante situaciones de este tipo. La sincronización entre las múltiples partes de un programa se llevaba a cabo mediante un bucle de suceso único. Estos entornos son de tipo síncrono, gestionados por sucesos. Entornos tales como el de Macintosh de Apple, Windows de Microsoft y X11/Motif fueron diseñados en torno al modelo de bucle de suceso.

Programas de flujo múltiple

En la aplicación de saludo, no se ve el hilo de ejecución que corre el programa. Sin embargo, Java posibilita la creación y control de hilos de ejecución explícitamente. La utilización de hilos (threads) en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar hilos de ejecución, permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se puede con otros lenguajes de tercera generación.

En un lenguaje orientado a Internet como es Java, esta herramienta es vital.

Si se ha utilizado un navegador con soporte Java, ya se habrá visto el uso de múltiples hilos en Java. Habrá observado que dos applets se pueden ejecutar al mismo tiempo, o que puede desplazar la página del navegador mientras el applet continúa ejecutándose. Esto no significa que el applet utilice múltiples hiloss, sino que el navegador es multihilo, multihilvanado o multithreaded.

Los navegadores utilizan diferentes hilos ejecutándose en paralelo para realizar varias tareas, "aparentemente" concurrentemente.

Por ejemplo, en muchas páginas web, se puede desplazar la página e ir leyendo el texto antes de que todas las imágenes estén presentes en la pantalla. En este caso, el navegador está trayéndose las imágenes en un hilo de ejecución y soportando el desplazamiento de la página en otro hilo diferente.

Las aplicaciones (y applets) multihilo utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo de ejecución para cada subtarea. Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multihilo permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

Vamos a modificar el programa de saludo creando tres hilos de ejecución individuales, que imprimen cada uno de ellos su propio mensaje de saludo, MultiHola.java:

```
// Definimos unos sencillos hilos. Se detendrán un rato
// antes de imprimir sus nombres y retardos
class TestTh extends Thread {
private String nombre;
private int retardo;
// Constructor para almacenar nuestro nombre
// y el retardo
public TestTh( String s,int d ) {
nombre = s;
retardo = d;
}
// El metodo run() es similar al main(), pero para
// threads. Cuando run() termina el thread muere
public void run() {
// Retasamos la ejecución el tiempo especificado
```

```
try {
sleep( retardo );
} catch( InterruptedException e ) {
;
}
// Ahora imprimimos el nombre
System.out.println( "Hola Mundo! "+nombre+" "+retardo );
}
}
public class MultiHola {
public static void main( String args[] ) {
TestTh t1,t2,t3;
// Creamos los threads
t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );
// Arrancamos los threads
t1.start();
t2.start();
t3.start();
}
}
```

Creación y Control de Hilos

Antes de entrar en más profundidades en los hilos de ejecución, se propone una referencia rápida de la clase Thread.

La clase Thread

Es la clase que encapsula todo el control necesario sobre los hilos de ejecución (threads).
Hay que distinguir claramente un objeto Thread de un hilo de ejecución o thread. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto Thread como el panel de control de un hilo de ejecución (thread). La clase Thread es la única forma de controlar el comportamiento de los hilos y para ello se sirve de los métodos que se exponen en las secciones siguientes.

Métodos de Clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase Thread. currentThread()

Este método devuelve el objeto thread que representa al hilo de ejecución que se está ejecutando actualmente.

yield()

Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que nos hilos de menor prioridad no sufran inanición.

sleep(long)

El método sleep() provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

Bibliografía Recomendada:

- A. Burns, A. Wellings: Concurrency in Ada. Cambridge University Press, 1998. *Fácil de leer, es una buena introducción a la concurrencia en Ada. Incluye temas que no se tocarán en el curso.*
- G. Andrews, F. Schneider: Concepts and Notations for Concurrent Programming. ACM Computing Surveys, vol. 15, n. 1, 1983, pp. 3-43. *Una revisión de conceptos y lenguajes para expresar concurrencia. Resume las propuestas más importantes en el área. Se dejará en fotocopiadora.*
- M. Ben-Ari: Principles of Concurrent Programming. Prentice-Hall, 1982. *Escueto, con poca orientación metodológica, pero con un contenido apreciable. Incluye soluciones a los problemas clásicos de concurrencia y un capítulo sobre Rendez-Vous en Ada.*

Adicional

- M. Ben-Ari: Ada for software engineers. John Wiley & Sons, 1998. *Una somera revisión de Ada, apropiada para alguien que conoce otros lenguajes y quiere introducirse en Ada, que incluye capítulos sobre concurrencia.*
- Gregory Andrews: Concurrent Programming, Principles and Practice. Benjamin Cummings, 1990. *Cubre casi todos los conceptos dados en la asignatura, más otros muchos relativos a algoritmos distribuidos, no necesarios en este nivel. Utiliza un lenguaje de programación propio.*
- Alan Burns, Geoff Davies: Concurrent programming. Addison-Wesley, 1993. *Una introducción a la concurrencia usando Pascal FCP. Adolece de una falta de metodología uniforme a la hora de afrontar los problemas.*
- N. H. Cohen: Ada as a Second Language. McGraw Hill. *El libro de referencia definitivo de Ada.*

Sitios consultados

- http://babel.ls.fi.upm.es/teaching/
- http://djaramillo2dani.blogspot.mx/2011/04/guia-practica-uno-1-estructura-228106.html
- http://marcelo-trabajo.blogspot.mx/
- http://cursos.aiu.edu/Lenguages%20de%20Programacion/PDF/Tema%201.pdf
- http://algoritmosylenguajes.blogspot.mx/2008/05/unidad-iii_31.html