UNIVERSIDAD AMERICANA

Programación Concurrente

Recopilación de teoría referente a la materia

Ing. Luis Müller

2011

Esta es una recopilación de la teoría referente a la asignatura Programación Concurrente, a ser estudiada en clases con los alumnos, y que servirá como base para su aplicación también en clases en ejercicios prácticos en el lenguaje de programación Java.

Contenido

Programación concurrente

RESUMEN

En diversos procesos computacionales es necesario aplicar diversos paradigmas de programación para enfocar el problema, es así que en algunas aplicaciones se necesita aplicar concurrencia y paralelismos para aumentar la eficiencia y potencial de los programas. En este contexto se va a desarrollar los principales conceptos de la programación concurrente y paralela, así como una introducción a la comunicación entre procesos y teoría de hilos.

Así mismo se presentan los ejemplos clásicos en donde se aplican. Además se presentara una introducción a la programación paralela, dando los alcances teóricos fundamentales. Finalmente se presentan las conclusiones obtenidas

INTRODUCCIÓN

Se conoce por programación concurrente a la rama de la informática que trata de las notaciones y técnicas de programación que se usan para expresar el paralelismo potencial entre tareas y para resolver los problemas de comunicación y sincronización entre procesos.

Actualmente observamos que el paradigma orientado a objetos, solo podemos ejecutar un equipo a la vez como máximo en cambio con la introducción de hilos 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.

Un ejemplo de programa concurrente

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 los diversos hilos involucradas en la descarga de todos los aspectos de una página. Los diversos hilos 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 botón **stop** como una suerte de conclusión se observa en este proceso que las hebras operan para llevar a cabo una tarea como la del ejemplo anterior así mismo se verá que los procesos deben tener acceso exclusivo a un recurso compartido como por ejemplo la visualización para evitar interferir unas con otras.

CONCEPTOS

Computación concurrente

La **computación concurrente** es la simultaneidad en la ejecución de múltiples tareas interactivas. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en un sola unidad central de proceso (multiprogramación), en varios procesadores o en una red de computadores distribuidos. La programación concurrente está relacionada con la programación paralela, pero enfatiza más la interacción entre tareas. Así, la correcta secuencia de interacciones o comunicaciones entre los procesos y el acceso coordinado de recursos que se comparten por todos los procesos o tareas son las claves de esta disciplina.

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

Los procesos secuenciales se han usado tradicionalmente para estudiar la concurrencia.

Tanto sistemas paralelos como distribuidos son concurrentes; pero un sistema concurrente puede ser no paralelo ni distribuido, como acontece, por ejemplo, con los sistemas operativos mono procesadores y multitarea, que son concurrentes pero no son paralelos ni distribuidos.

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.

Un proceso pasa por una serie de datos discretos. Se dice que un proceso se está ejecutando (estado de ejecución), si tiene asignada la CPU. Se dice que un proceso está listo (estado listo) si pudiera utilizar la CPU en caso de haber una disponible. Un proceso está bloqueado (estado bloqueado) si se está esperando que suceda algún evento antes de poder seguir la ejecución.

Como ya se menciono los procesos son concurrentes si existen simultáneamente. La concurrencia es el punto clave de la Multiprogramación, el Multiproceso y el Proceso distribuido y fundamental para el diseño de sistemas operativos. La concurrencia comprende un gran número de cuestiones diseño, incluyendo la comunicación entre procesos, compartición y competencia por lo recursos, sincronización de la ejecución de varios procesos y asignación del tiempo de procesador a los procesos.

PROGRAMA CONCURRENTE: Es un programa diseñado para tener 2 o más contextos de ejecución decimos que este tipo de programa es multi-hilo, porque tiene más de un contexto de ejecución.

Un programa concurrente es un programa que tiene más de una línea lógica de ejecución, es decir, es un programa que parece que varias partes del mismo se ejecutan simultáneamente. Un ejemplo de de esto es un programa que realice determinada función y, simultáneamente, exponga datos en la pantalla. Un programa concurrente puede correr en varios procesadores simultáneamente o no. Esta importancia de la concurrencia es especialmente destacable en sistemas operativos como Linux, que además de concurrentes, presentan unos mecanismos de concurrencia estables.

La concurrencia puede presentarse en tres contextos diferentes:

- **Varias aplicaciones**: La multiprogramación se creó para permitir que el tiempo de procesador de la maquina fuese compartido dinámicamente entre varios trabajos o aplicaciones activas.
- **Aplicaciones estructuradas**: Como aplicación de los principios del diseño modular y la programación estructurada, algunas aplicaciones pueden implementarse eficazmente como un conjunto de procesos concurrentes.
- **Estructura del Sistema Operativo**: Las mismas ventajas de estructuración son aplicables a los programadores de sistema y se han comprobado que algunos sistemas operativos están implementados con un conjunto de procesos.

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

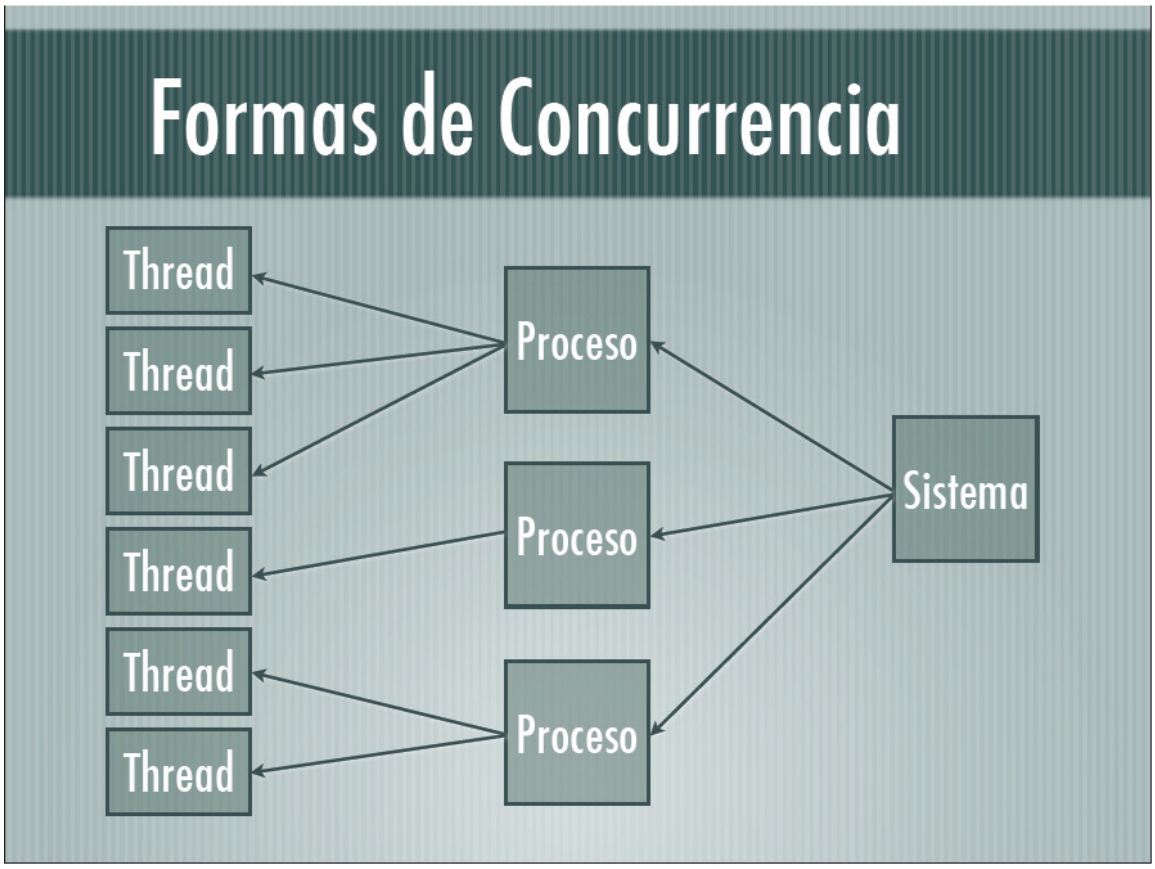
PROGRAMA DISTRIBUIDO: Es un sistema diseñado para ejecutarse simultáneamente en una red de procesadores autónomos, que no comparten la memoria principal, con cada programa en su procesador aparte.

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. Ej.:

En la edición de documentos o archivos 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 multi-hilo en el que alguno de los datos residen simultáneamente en cada contexto de ejecución.

FUNDAMENTOS DE LA PROGRAMACIÓN CONCURRENTE

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



Cuando dos o más procesos llegan al mismo tiempo a ejecutarse, se dice que se ha presentado una concurrencia de procesos. Es importante mencionar que para que dos o más procesos sean concurrentes, es necesario que tengan alguna relación entre ellos como puede ser la cooperación para un determinado trabajo o el uso de información o recursos compartidos, por ejemplo: en un sistema de un procesador, la multiprogramación es una condición necesaria pero no suficiente para que exista concurrencia, ya que los procesos pueden ejecutarse de forma totalmente independiente.

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 cálculo 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 un hilo (PROCESO):

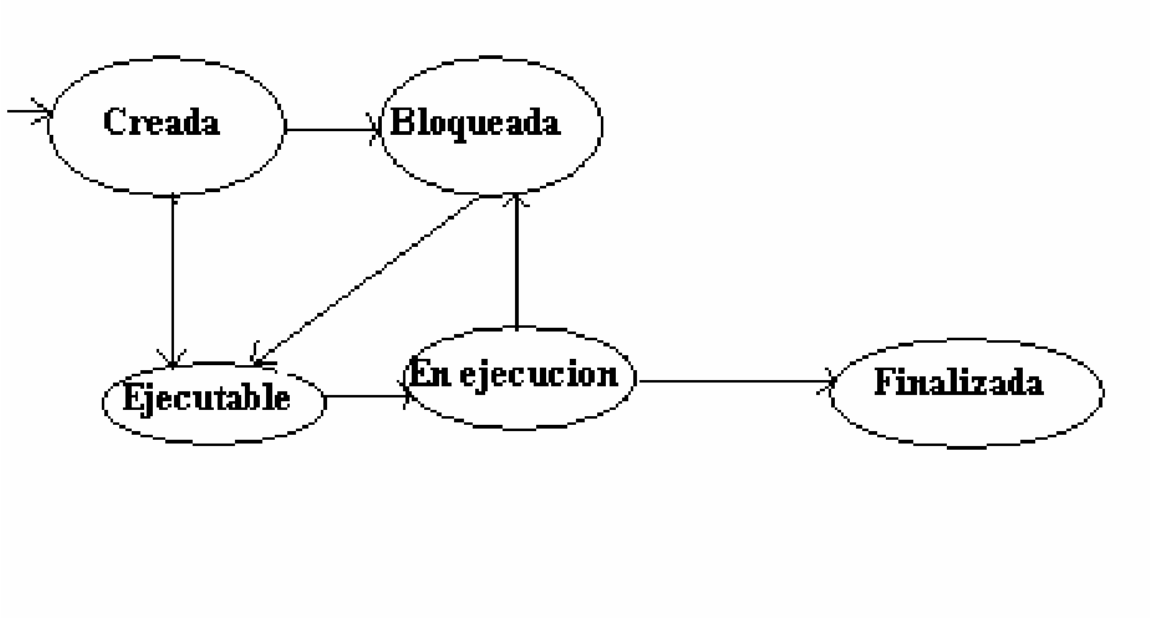
1. **Creada**: El hilo se ha creado pero aun no está listo para ejecutarse.

2. **Ejecutable o lista**: El hilo está listo para ejecutarse pero espera a conseguir un procesador en que ejecutarse.

3. **En ejecución**: El hilo se está ejecutando en un procesador.

4. **Bloqueado o en espera**: El hilo está esperando tener acceso a una sección crítica o ha dejado el procesador voluntariamente.

5. **Finalizada**: El hilo se ha parado y no se volverá a ejecutar.



Análisis de la comunicación entre procesos

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

- Los hilos (incluso los procesos), compiten por un acceso exclusivo a los recursos compartidos, como los archivos físicos: archivos o datos.

- Los hilos se comunican para intercambiar datos.

En ambos casos es necesario que los hilos sincronicen su ejecución para evitar conflictos cuando adquieren los recursos, o para hacer contacto cuando intercambian datos. Un hilo puede comunicarse con otros mediante:

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

- 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.

Esta situación se plantea, por ejemplo, en un problema típico de la programación concurrente conocido como el Problema de los Jardines.

Así para garantizar la exclusión mutua tenemos las siguientes opciones:

Emplear variables de cerradura: Consiste en poner una variable compartida, una cerradura, a 1 cuando se va a entrar en la región critica, y devolverla al valor 0 a la salida. Esta solución en sí misma no es válida porque la propia cerradura es una variable crítica. La cerradura puede estar a 0 y ser comprobada por un proceso A, este ser suspendido, mientras un proceso B chequea la cerradura, la pone a 1 y puede entrar a su región critica; a continuación A la pone a 1 también, y tanto A como B pueden encontrar en su sección critica al mismo tiempo.

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 porque 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:

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

- incrementar el valor del registro

- 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.

Para evitar errores se pueden identificar aquellas regiones de los procesos que acceden a variables compartidas y dotarlas de la posibilidad de ejecución como si fueran una única instrucción.

Sección Crítica

Se denomina **Sección Crítica** a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción. Esto quiere decir que si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables compartidas, entonces otro proceso no puede entrar a ejecutar una región crítica en la que acceda a variables compartidas con el anterior.

Las secciones críticas se pueden agrupar en clases, siendo **mutuamente exclusivas** las secciones críticas de cada clase. Para conseguir dicha exclusión se deben implementar protocolos software que impidan o **bloqueen** (lock) el acceso a una sección crítica mientras está siendo utilizada por un proceso.

La exclusión mutua necesita ser aplicada solo cuando un proceso acceda a datos compartidos; cuando los procesos ejecutan operaciones que no estén en conflicto entre sí, debe permitírseles proceder de forma concurrente. Cuando un proceso esta accesando datos se dice que el proceso se encuentra en su sección critica (o región critica).

Mientras un proceso se encuentre en su sección crítica, los demás procesos pueden continuar su ejecución fuera de sus secciones críticas. Cuando un proceso abandona su sección critica, entonces debe permitírsele proceder a otros procesos que esperan entrar en su propia sección crítica (si hubiera un proceso en espera). La aplicación de la exclusión mutual es uno de los problemas clave de la programación concurrente. Se han diseñado muchas soluciones para esto: algunas de software y algunas de hardware, más de bajo nivel y otras de alto nivel; algunas que requieren de cooperación voluntaria, y algunas que demandan una adherencia rígida a protocolos estrictos.

Estar dentro de una sección crítica es un estado muy especial asignado a un estado. El proceso tiene acceso exclusivo a los datos compartidos, y todos los demás procesos que necesitan accesar a esos datos permanecen en espera. Por tanto, las secciones críticas deben ser ejecutadas lo más rápido posible, un programa no debe bloquearse dentro de su sección crítica, y las secciones críticas deben ser codificadas con todo cuidado.

Si un proceso dentro de una sección crítica termina, tanto de forma voluntaria como involuntaria, entonces, al realizar su limpieza de terminación, el sistema operativo debe liberar la exclusión mutua para que otros procesos puedan entrar en sus secciones críticas

BLOQUEO MEDIANTE EL USO DE VARIABLES COMPARTIDAS

Vamos a intentar una forma de implementar el bloqueo a una sección crítica mediante el uso de una variable compartida que se suele denominar indicador (flag). Iremos realizando una serie de refinamientos del algoritmo que servirán para ir poniendo de manifiesto el tipo de problemas que suelen presentarse en la implementación de la concurrencia entre procesos.

Asociaremos a un recurso que se comparte un flag. Antes de acceder al recurso, un proceso debe examinar el indicador asociado que podrá tomar dos valores (True o False) que indicarán, de forma respectiva, que el recurso está siendo utilizado o que está disponible. El ejemplo muestra el uso de un indicador para implementar la exclusión mutua entre dos procesos.

El mecanismo de bloqueo y de desbloqueo se implementa mediante dos procedimientos. Para bloquear el acceso a la región crítica ambos procesos llaman al procedimiento de bloqueo pero con los indicadores en distinto orden. La solución tiene el inconveniente de que durante la espera de la liberación del recurso, el proceso permanece ocupado (busy wait). Pero hay un problema mayor. Supongamos que ambos procesos realizan la llamada al bloqueo de forma simultánea. Cada proceso puede poner su propio indicador y comprobar el estado del otro. Ambos verán los indicadores contrarios como ocupados y permanecerán a la espera de que el recurso quede liberado, pero esto no podrá suceder al no poder entrar ninguno en su sección crítica. Esta acción se conoce como **interbloqueo** (deadlock).

El error más serio que puede ocurrir en entornos concurrentes es el conocido como **interbloqueo,** que consiste en que dos o más procesos entran en un estado que imposibilita a cualquiera de ellos salir del estado en que se encuentra. A dicha situación se llega porque cada proceso adquiere algún recurso necesario para su operación a la vez que espera a que se liberen otros recursos que retienen otros procesos, llegándose a una situación que hace imposible que ninguno de ellos pueda continuar.

El interbloqueo se produce porque la desactivación del indicador asociado a un proceso se produce una vez que se ha completado el acceso a la sección crítica. Se puede intentar resolver el problema haciendo que el proceso desactive su propio indicador durante la fase de bloqueo siempre que encuentre que el indicador del otro proceso está activado.

CARACTERIZACIÓN DEL INTERBLOQUEO

Para que se dé una situación de interbloqueo se deben cumplir de forma simultánea las cuatro condiciones siguientes.

Exclusión mutua. Los procesos utilizan de forma exclusiva los recursos que han adquirido. Si otro proceso pide el recurso debe esperar a que este sea liberado.

Retención y espera. Los procesos retienen los procesos que han adquirido mientras esperan a adquirir otros recursos que está siendo retenido por otros procesos.

No existencia de expropiación. Los recursos no se pueden quitar a los procesos que los tienen; su liberación se produce voluntariamente una vez que los procesos han finalizado su tarea con ellos.

Espera circular. Existe una cadena circular de procesos en la que cada proceso retiene al menos un recurso que es solicitado por el siguiente proceso de la cadena.

La condición de espera circular implica la condición de retención y espera, sin embargo, resulta útil considerar ambas condiciones por separado.

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:

- Inicializar

- Espera (wait)

- Señal (signal)

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.

Las operaciones pueden describirse como sigue:

• **inicializa** (S: SemaforoBinario; v: integer)

Poner el valor del semáforo S al valor de v (0 o 1)

• **espera** (S)

if S = 1 **then** S := 0

else suspender la tarea que hace la llamada y ponerla en la cola de tareas

• **señal** (S)

if la cola de tareas está vacía **then** S := 1

else reanudar la primera tarea de la cola de tareas

La operación **inicializa** se debe llevar a cabo antes de que comience la ejecución concurrente de los procesos ya que su función exclusiva es dar un valor inicial al semáforo.

Un proceso que corre la operación **espera** y encuentra el semáforo a 1, lo pone a 0 y prosigue su ejecución. Si el semáforo está a 0 el proceso queda en estado de espera hasta que el semáforo se libera. Dicho estado se debe considerar como uno más de los posibles de un proceso. Esto es así debido a que un proceso en espera de un semáforo no está en ejecución ni listo para pasar a dicho estado puesto que no tiene la CPU ni puede pasar a tenerla mientras que no se lo indique el semáforo.

El ejemplo que sigue muestra la solución que el uso de semáforos ofrece al problema clásico del **productor-consumidor**. Este problema aparece en distintos lugares de un sistema operativo y caracteriza a aquellos problemas en los que existe un conjunto de procesos que producen información que otros procesos consumen, siendo diferentes las velocidades de producción y consumo de la información. Este desajuste en las velocidades, hace necesario que se establezca una sincronización entre los procesos de manera que la información no se pierda ni se duplique, consumiéndose en el orden en que es producida.

Ejemplo productor-consumidor

Suponemos que se tienen dos procesos P1 y P2; P1 produce unos datos que consume P2. P1 almacena datos en algún sitio hasta que P2 está listo para usarlos. Por ejemplo, P1 genera información con destino a una impresora y P2 es el proceso gestor de la impresora que se encarga de imprimirlos. Para almacenar los datos se dispone de un buffer o zona de memoria común al productor y al consumidor. Se supone que para almacenar y tomar datos se dispone de las funciones Poner(x) y Tomar(x). Para saber el estado del buffer se usan las funciones Lleno, que devuelve el valor TRUE si el buffer está lleno, y Vacio, que devuelve el valor TRUE si el buffer está vacío. El productor y el consumidor corren a velocidades distintas, de modo que si el consumidor opera lentamente y el buffer está lleno, el productor deberá bloquearse en espera de que haya sitio para dejar más datos; por el contrario si es el productor el que actúa lentamente, el consumidor deberá bloquearse si el buffer está vacío en espera de nuevos datos.

Interbloqueo e Injusticia

Se dice que un hilo está en estado de ínter bloqueo, si está esperando un evento que no se producirá nunca, el ínter bloqueo implica varios hilos, cada uno de ellos a la espera de recursos existentes en otros. Un ínter bloqueo puede producirse, siempre que dos o más hilos compiten por recursos; para que existan ínter bloqueos son necesarias 4 condiciones:

- Los hilos deben reclamar derechos exclusivos a los recursos.

- Los hilos deben contener algún recurso mientras esperan otros; es decir, adquieren los recursos poco a poco, en vez de todos a la vez.

- No se pueden sacar recursos de hilos que están a la espera (no hay derecho preferente).

- Existe una cadena circular de hilos en la que cada hilo contiene uno o más recursos necesarios para el siguiente hilo 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 barrera 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.

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 la cola de la variable de condición utilizada. Si no hay ningún proceso suspendido en la cola de la variable de condición invocada la operación señal no tiene ningún efecto.

Mensajes

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:

- El modo de nombrar los procesos.

- El modelo de sincronización.

- Almacenamiento y estructura del mensaje.

ANALISIS DE CONCEPTOS MEDIANTE 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 esté 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 bloqueo"[7].

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

Decimos que un hilo esta aplazado indefinidamente si se retrasa 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 está 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? [7].

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 último 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 más 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 sería: 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.

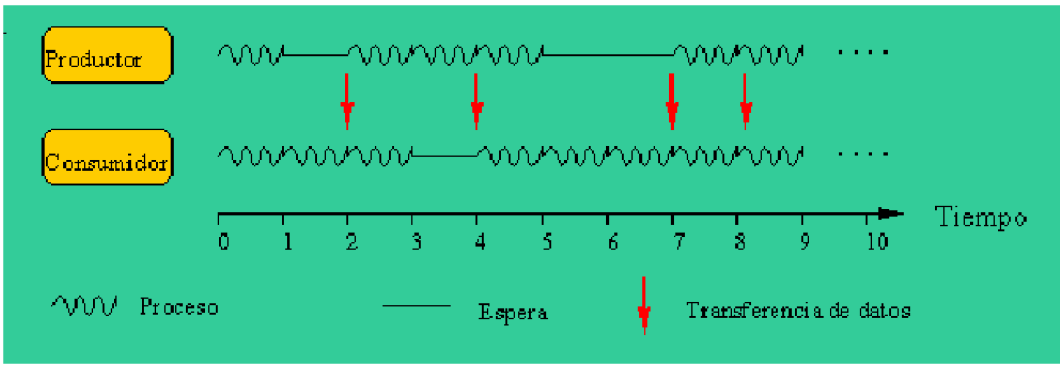
Luego verifica entonces si el número de clientes que esperan es menor que el número de sillas, si esto no ocurre, sale sin su corte de pelo. Si existe una silla disponible, el cliente incrementa una variable contadora. Luego realiza un up en el semáforo “customer” con lo que despierta al barbero. En este momento tanto el cliente como el barbero están despiertos, luego cuando le toca su turno al cliente le cortan el pelo[7].

El problema del productor-consumidor.

Estudiaremos problemas en la comunicación, supongamos dos tipos de procesos: Productores y Consumidores.

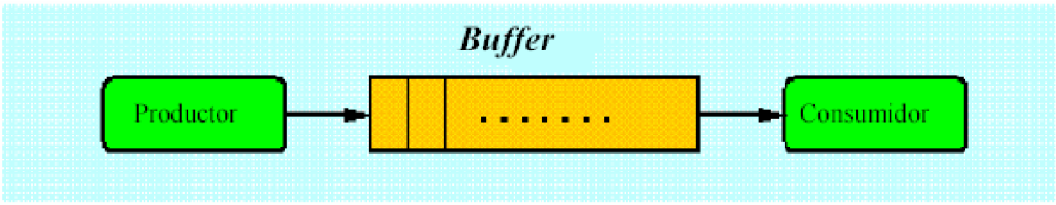
Productores: Procesos que crean elementos de datos mediante un procedimiento interno (Produce), estos datos deben ser enviados a los consumidores.

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.

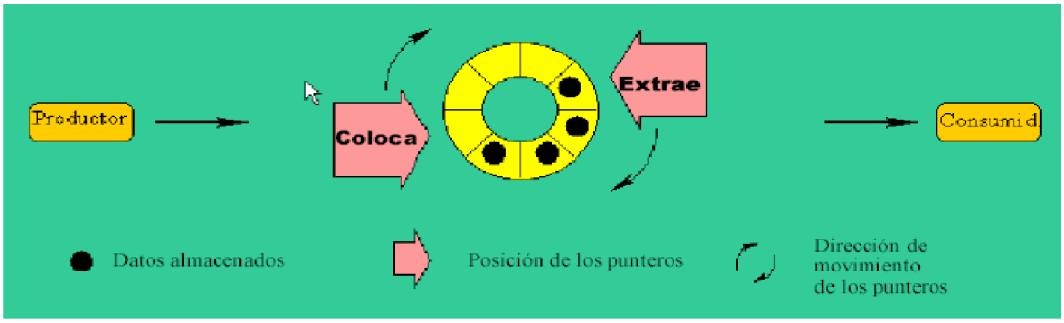


Problema: El productor envía un dato cada vez, y el consumidor consume un dato cada vez. Si uno de los dos procesos no está listo, el otro debe esperar.

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

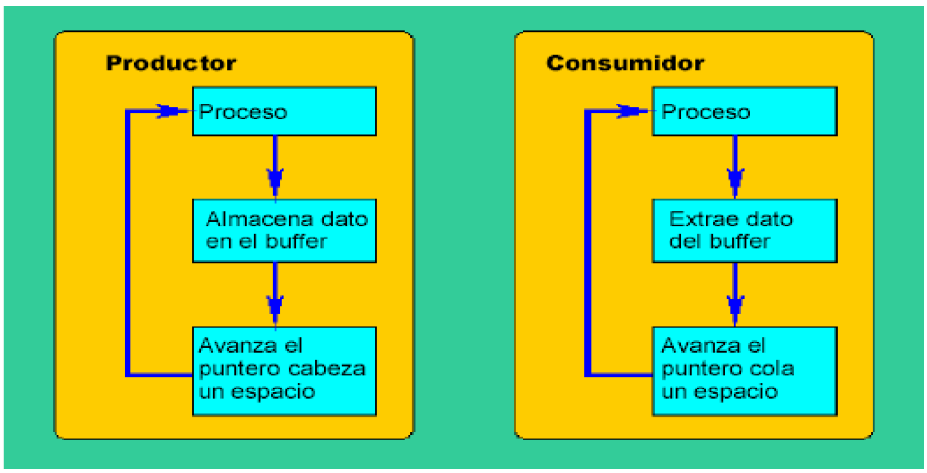


El buffer puede ser infinito. No obstante esto no es realista



Alternativa: Buffer acotado en cola circular

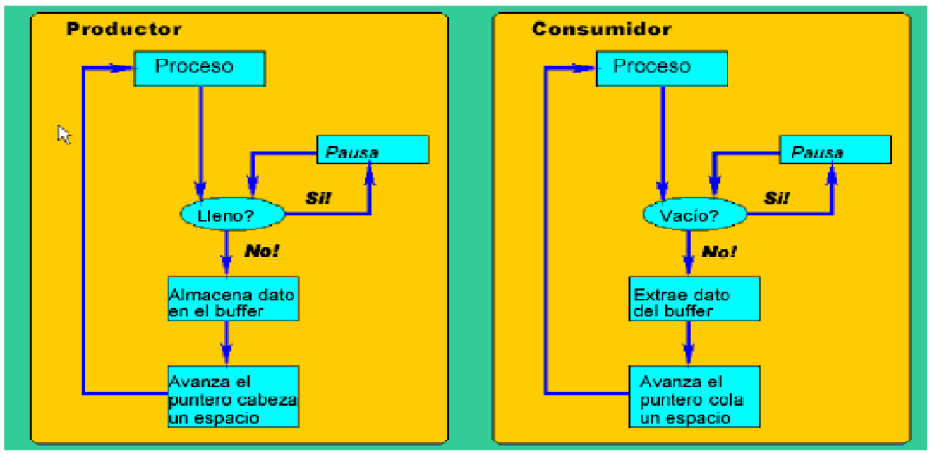
Algoritmo de funcionamiento del buffer acotado



Problemas:

- El productor puede enviar un dato a un buffer lleno
- 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?

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

- Necesitamos dos condiciones: lleno y vacío.

- 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.

- 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.

CARACTERISTICAS DE LOS PROCESOS CONCURRENTES:

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

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

- 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.

Las interacciones entre procesos tienen dos formas:

- **Primero comunicación**.- Si tenemos varios procesos ejecutando y queremos que interactúen entre ellos para solucionar un determinado problema, y por tanto, necesitan intercambiar información, es decir necesitan comunicarse. Esto implica el intercambio de datos entre procesos, ya sea por medio de un mensaje implícito o a través de variables compartidas. Una variable es compartida entre procesos si es visible al código de esos procesos.
  Mediante la comunicación se consigue que la ejecución de un proceso influya en la ejecución de otro.
- **Segundo sincronización**.- La sincronización (sinónimo de espera) es habitualmente necesaria cuando dos procesos se comunican puesto que la velocidad a la que se ejecutan no es predecible. Para que un proceso se comunique con otro, aquel deberá realizar acciones que sea capaz de detectar y esto solo funciona si realizar y detectar ocurren en este orden. Relaciona el seguimiento de un proceso con el seguimiento de otro. En otras palabras, la sincronización implica intercambio de información de control entre procesos.

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 esté disponible.

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 está 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:

- Los procesos necesitan acceso exclusivo a los recursos.

- Los procesos necesitan mantener ciertos recursos exclusivos y otros en espera.

- Los recursos no se pueden obtener de los procesos que están a la espera.

PROBLEMAS DE LA CONCURRENCIA.

En los sistemas de tiempo compartido (aquellos con varios usuarios, procesos, tareas, trabajos que reparten el uso de CPU entre estos) se presentan muchos problemas debido a que los procesos compiten por los recursos del sistema. Los programas concurrentes a diferencia de los programas secuenciales tienen una serie de problemas muy particulares derivados de las características de la concurrencia:

1. **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 del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua.
2. **Bloqueo mutuo o deadlock**: Un proceso se encuentra en estado de deadlock si está esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir deadlock:

- Los procesos necesitan acceso exclusivo a los recursos.
- Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
- Los recursos no se pueden obtener de los procesos que están a la espera.
- Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena.

3. **Retraso indefinido o starvation**: Un proceso se encuentra en starvation si es retrasado indefinidamente esperando un suceso que no puede ocurrir. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso.

PROGRAMACIÓN PARALELA

Computación distribuida

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 clúster 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.

Necesidad de la programación paralela

- Resolver problemas que no caben en una CPU.

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

- Se puede ejecutar :

    - Problemas mayores.

    - Problemas más rápidamente.

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

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

Aspectos en la programación paralela

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

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

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

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

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

Es habitual encontrar en la bibliografía el término del *programa concurrente* en el mismo contexto que el *programa paralelo o distribuido*. Existen diferencias sutiles entre estos dos conceptos:

- **Programación concurrente**: Es aquel que define acciones que pueden realizarse simultáneamente.
- **Programación paralelo**: Es un programa concurrente diseñado para su ejecución en un hardware paralelo.
- **Programa distribuido**: Es un programa paralelo diseñado para su ejecución en una red de procesadores autónomos que no comparten la memoria.

Preguntas:

¿Qué es concurrencia?

- Habilidad para ejecutar varias actividades en paralelo o Simultáneamente

Las máquinas monoprocesador/mono tarea se quedaban sin hacer nada mientras se realizaban operaciones de E/S

Con la concurrencia, mientras ocurre E/S, se quiere aprovechar la CPU para otros cálculos

Ejemplos de Multitarea

- Nula: Sin multitarea (Ej.: MS-DOS)

- Cooperativa: Los procesos deben ceder la CPU a intervalos

- regulares (Ej.: Windows 3.11) ¿Problema?

- Preferente: El SO administra los procesadores. Cada proceso tiene un tiempo T para usar la CPU (Ej.: Unix, FreeBSD, Linux, WinNT, etc.)

- Real: Preferente pero con varios procesadores

Motivaciones

- Utilizar plenamente el procesador Mientras E/S, realizar otras tareas

- Aprovechar el paralelismo real si lo hubiera

- Modelar el paralelismo existente en el mundo real

Paralelismo

Procesamiento físicamente simultáneo

Necesita múltiples elementos de procesamiento

Ej.: procesos corriendo en una máquina multiprocesador

PROGRAMACION EN JAVA

La programación concurrente se basa en construir programas que se ejecutan en paralelo. Mediante comunicación y sincronización estos programas cooperan para realizar una tarea en conjunto. Ejemplos típicos son las interfaces gráficas, las bases de datos distribuidas y las simulaciones a gran escala.

Dentro de los posibles modelos de concurrencia (Bal et. al. 1989), el modelo de procesos secuenciales compartiendo memoria - thread model, es uno de los más utilizados, en parte porque las tecnologías de lenguajes de programación (Java), de sistemas operativos (Windows, Linux) y de arquitecturas de procesador, ponen a los threads al tope de sus prioridades.

Hilos de java:

Un hilo puede estar en uno de estos cinco estados: creado, ejecutable, en ejecución, bloqueado o finalizado. Un hilo 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, un hilo es una clase por tanto la manera más sencilla de crear un hilo 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 un hilo creado sea ejecutable, llamamos sencillamente a su método start:

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

Después de algún coste adicional, el hilo recientemente ejecutable transfiere el control a su método run cada clase que amplía 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 un hilo contiene un bucle, ya que la salida del método run finaliza el hilo. 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 el hilo del estado en ejecución a un estado bloqueado, donde espera una interrupción del temporizador 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 el hilo salga de su método run, lo que después de un retraso para la limpieza finalizaría el hilo, también se puede finalizar un hilo, 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 que el hilo utiliza para continuar el bucle por ejemplo:

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

Para detener el hilo anterior, otro hilo 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 interacción extra del bucle.

A veces no es conveniente hacer sus clases de la clase thread , por ejemplo, podemos querer que nuestra clase applet sea un hilo separado. 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.

SINCRONIZACIÓN EN 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 los hilos separados . una manera de llevar esto a cabo seria pasar al objeto compartido, como parámetro del constructor de cada hilo. En el caso de una variable compartida de productor-consumidor, esto podría quedar así:

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

Donde damos por supuesto que tanto producer And Consumer amplían switch.

REFERENCIAS.-

- Programación concurrente, Fernando cuartero, octubre del 2001.
- Programación concurrente con java, Juan pavón mestras, Dep. sistemas informáticos y programación, universidad complutense de Madrid.
- Programación paralela y concurrente, M enc., Mario Farias Elvis.
- Programación Concurrente, introducción a los hilos, José Toribio Vicente. Barcelona , España 7/11/200.
- Herramientas para programación paralela, José Manuel García Carrasco. Departamento de Informática, Universidad de Castilla La mancha.
- Lenguajes de programación, principios y paradigmas, A. Tucker R. Noonan, editorial MC graw Hill 2003.
- Sistemas operativos modernos, Andrew S. Tanehbaum, Prentice- may hispanoamericana 1992.
- http://itver.edu.mx/so1/definiciones%20de%20proceso.htm

  El termino “proceso”, fue utilizado por primera vez por los diseñadores del sistema Multics en los años 60`s.
- http://lara.pue.udlap.mx/sist_oper/capitulo5.html

  En los sistemas de tiempo compartido se presentan muchos problemas debido a los procesos que compiten por algún recurso. Aqui se muestran algunos de los problemas de Concurrencia.
- http://agamenon.uniandes.edu.co/~c21437/ma-bedoy/taxonomi.htm

  Taxonomia de las arquitecturas de computadoras, la cual se le debe a Flynn.
- http://itver.edu.mx/so1/Definiciones%2032.htm

  Concurrencia de procesos. Definicion de concurrencia.
- http://1ml.ls.fi.upm.es/~angel/concurrente/96notes/node15.html

  Estado de un programa concurrente
- http://www.biocristalografria.df.ibilce.unesp.br/irbis/disertacion/node239.html

  Concepto de concurrencia
- http://gsyc.escet.urjc.es/simple_com/phd-thesis-es/node53.html

  Ejecucion de un programa concurrente
- http://1ml.ls.fi.upm.es/~angel/concurrente/96notes/node16.html

  Concepto de sincronización
- http://tiny.uasnet.mx/prof/cln/ccu/mario/sisop/sisop06.htm

  Concepto de monitores.

- http://fciencias.ens.uabc.mx/notas_cursos/so2/c_concu.html

  C concurrente.

- http://www.geocities.com/SiliconValley/Bay/9418/2_CONCEPTOS.htm

  Conceptos basicos de paralelismo y concurrencia.

- http://research.cem.itesm.mx/jesus/cursos/compd2/s2/node2.html

  Caracteristicas de los lengiajes paralelos.

  -Paralelismo explicito

  -Paralelismo implícito

- http://www.inf.ufrgs.br/procpar/tools/cc++exeCp.htm

  Ejemplo de C paralelo.