1. INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE Y PARALELA

1.1. HISTORIA DE LA PROGRAMACIÓN CONCURRENTE

En los tiempos actuales la construcción de sistemas paralelos y concurrentes tiene cada vez menos limitantes y su existencia ha hecho posible obtener gran eficiencia en el procesamiento de datos. La utilización de tales sistemas se ha extendido a muchas áreas de la Ciencia Computacional e Ingeniería. Como es sabido, existen infinidad de aplicaciones que tratan de obtener el máximo rendimiento del sistema al resolver un problema utilizando máquinas con un solo procesador; sin embargo, cuando el sistema no puede proporcionar el rendimiento esperado, una posible solución es optar por aplicaciones, lenguajes de programación, arquitecturas e infraestructuras de procesamiento paralelo y concurrente. Para incrementar el rendimiento de determinados sistemas, el procesamiento paralelo es, por tanto, una alternativa al procesamiento secuencial. Desde el punto de vista práctico, hoy en día existen arquitecturas de computadores implementadas en sistemas reales que proporcionan auténtico procesamiento paralelo que justifican el interés actual en llevar a cabo investigaciones dentro del procesamiento paralelo y áreas a fines (concurrencia, sistemas distribuidos, sistemas de tiempo real, etc.). Una parte importante de estas investigaciones se refiere a mejorar el diseño de algoritmos paralelos y distribuidos, el desarrollo de metodologías y modelos de programación paralela que ayuden a la programación de estos algoritmos sobre sistemas físicamente distribuidos, así como herramientas de razonamiento que permitan garantizar su corrección.

Procesamiento Paralelo

El procesamiento paralelo ha provocado un tremendo impacto en muchas áreas donde las aplicaciones computacionales tienen cabida. Un gran número de aplicaciones computacionales en la ciencia, ingeniería, comercio o medicina requieren de una alta velocidad de cálculo para resolver problemas de visualización, bases de datos distribuidas, simulaciones, predicciones científicas, etc.

Estas aplicaciones envuelven procesamiento de datos o ejecución de un largo número de iteraciones de cálculo. El procesamiento paralelo es uno de los enfoques computacionales que hoy en día puede ayudarnos a procesar estos cálculos de una manera más viable. Este incluye el estudio de arquitecturas paralelas y algoritmos paralelos.

Las tecnologías del procesamiento paralelo son actualmente explotadas de forma comercial, principalmente para el trabajo en el desarrollo de herramientas y ambientes. El desarrollo de esta área en las redes de computadoras ha dado lugar a la llamada Computación Heterogénea [Bacci et al.99] (que proporciona un ambiente

donde la aplicación paralela es ejecutada utilizando diferentes computadoras secuenciales y paralelas en las cuales la comunicación se lleva a cabo a través de una red inteligente. Este enfoque proporciona alto rendimiento).

Existen muchos factores que contribuyen al rendimiento de un sistema paralelo, entre ellos están la integración entre procesos, es decir, el tener múltiples procesos activos simultáneos resolviendo un problema dado; la arquitectura del hardware (arquitectura superescalar, de vector o pipelinizada, etc.), y enfoques de programación paralela para comunicar y sincronizar los procesos.

Paralelismo

El objetivo del paralelismo es conseguir ejecutar un programa inicialmente secuencial en menos tiempo utilizando para ello varios procesadores. La idea central es dividir un problema grande en varios más pequeños y repartirlos entre los procesadores disponibles, pero, sin una estrategia adecuada de diseño, dicho reparto no produce los resultados esperados respecto del aumento de eficiencia del programa, así como puede complicar enormemente la programación y el mantenimiento posterior. Así, a veces ocurre que no es posible llevar a cabo una paralelización de todo el programa, debido a que aparecen elementos que no se pueden paralelizar, otras veces el reparto de elementos del programa entre los procesadores no se hace de manera equitativa, lo que se traduce en un aumento muy pobre de la eficiencia respecto del tiempo de ejecución.

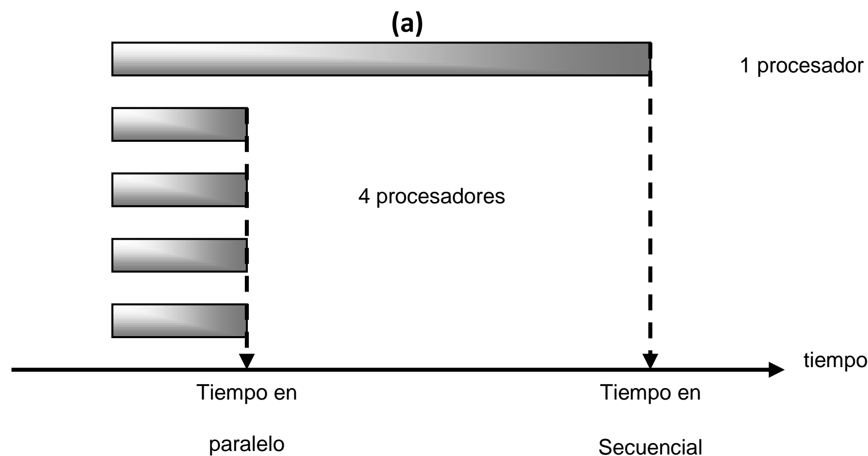


Fig. 1. (a) Esquema ideal del resultado de la paralelización

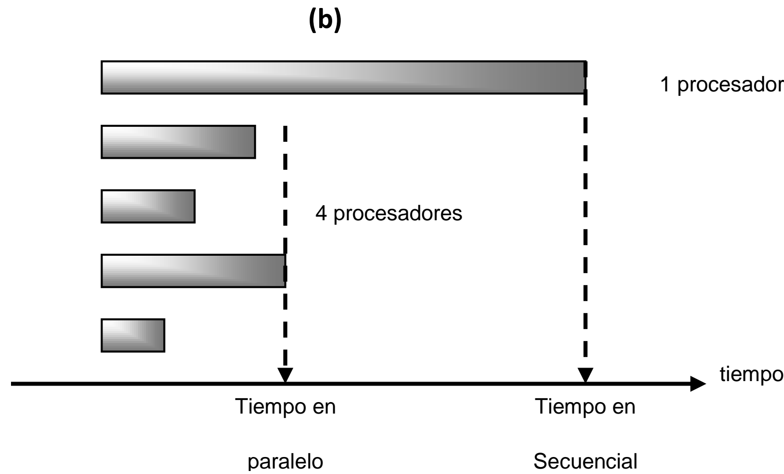


Fig. 1. (b) esquema real, del resultado de la paralelización

Si suponemos que todos los procesadores llevan a cabo el mismo tipo y número de operaciones, la reducción ideal del tiempo de ejecución debería ser la que se muestra en la fig. 1. (a); sin embargo, lo que realmente ocurre con frecuencia es lo mostrado en la fig. 1 (b).
Un ejemplo de paralelización puede ser un ciclo de 100 iteraciones que se reparte para su ejecución entre 4 procesadores. Cada uno de ellos sólo ejecutará 25 repeticiones del ciclo original, Fig. 2.

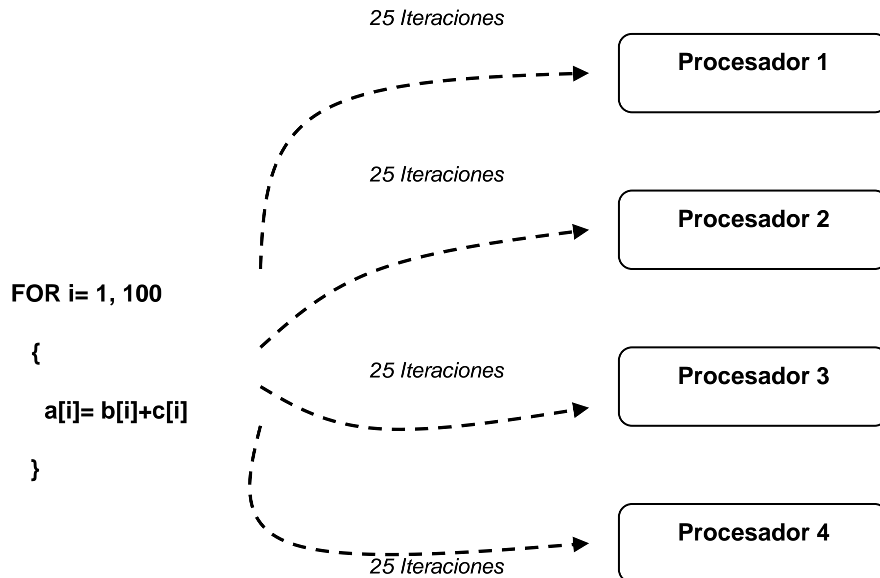


Fig. 2. Reparticiones de las iteraciones de un ciclo en 4 procesadores

Sin embargo, esto no suele funcionar de esta forma en el caso real, ya que suelen existir relaciones entre las variables de los programas que establecen dependencias secuenciales que impiden la realización de repartos simples, como en el caso anterior. Es por esto que la unidad de paralelización en los lenguajes de programación y sus aplicaciones no debiera ser las instrucciones de los programas, sino más bien entidades que encapsulen instrucciones así como los datos que modifican.

Concurrencia

La programación concurrente tiene sus raíces en los sistemas operativos y en la programación de sistemas, no en vano, los primeros programas concurrentes fueron los propios sistemas operativos de multiprogramación en los que un solo procesador de gran capacidad debía compartir su tiempo entre muchos usuarios. En los años 60's se introdujeron en las computadoras dispositivos controladores independientes de entrada-salida llamados canales. Estos canales eran programables en sí mismos. Los sistemas operativos fueron organizados como una colección de procesos ejecutándose concurrentemente ("al mismo tiempo"), algunos en los canales y otros ejecutándose en el procesador principal o CPU. Por otro lado en esos años la programación de sistemas con capacidades de concurrencia se hacía a bajo nivel, en ensamblador, pues aparte de no disponer de lenguajes de alto nivel con capacidades de concurrencia, se primaba la supuesta eficiencia del código escrito directamente en ensamblador. La aparición en 1972 del lenguaje de alto nivel Concurrent Pascal, desarrollado por Brinch Hansen abrió la puerta a otros lenguajes de alto nivel que incorporaban concurrencia.

Desde entonces la programación concurrente ha ido ganando interés y actualmente se utiliza muy a menudo en la implementación de numerosos sistemas. Tres grandes hitos fortalecen el hecho de que la programación concurrente sea tan importante:

- La aparición del concepto de Thread o hilo que hace que los programas puedan ejecutarse con mayor velocidad comparados con aquellos que utilizan el concepto de proceso
- La aparición más reciente de lenguajes como JAVA, lenguaje orientado a objetos de propósito general que da soporte directamente a la programación concurrente mediante la inclusión de primitivas específicas.
- La aparición del INTERNET que es un campo abonado para el desarrollo y la utilización de programas concurrentes. Cualquier programa de Internet en el que podamos pensar tales como un navegador, un chat, etc., están programados usando técnicas de programación concurrente.

1.2. LENGUAJES DE PROGRAMACIÓN CONCURRENTE

Los lenguajes de programación paralela/concurrente se basan en dos categorías, en abstracciones de programación concurrente basadas en exclusión mutua de accesos a una memoria individual; y en abstracciones de procesos que se comunican mediante el envío de mensajes unos con otros. El envío de mensajes es una acción de alto nivel que puede ser implementada físicamente mediante procesadores distribuidos.

Cada enfoque de programación concurrente/paralela sugiere una configuración de hardware particular. La mejor será aquella que empate con las primitivas del lenguaje utilizado en la programación paralela/concurrente. Actualmente existen muchos lenguajes de programación que ya tienen diseñadas primitivas para el manejo de procesos de manera asíncrona o síncrona. La programación asíncrona se utiliza para la programación de multiprocesadores o sistemas distribuidos; mientras que las soluciones paralelas síncronas son propias del uso de arrays o vectores de procesadores.

En el caso de la programación orientada a objetos, ésta puede ser utilizada como un buen enfoque de programación paralela/concurrente ya que puede encapsular y abstraer patrones comunes de comunicación paralela y llevar dicha abstracción hacia un estilo estructurado de programación paralela/concurrente.

El lenguaje de programación C++ es un ejemplo de un excelente lenguaje de programación orientado a objetos con el que se pueden implementar aplicaciones paralelas mediante el uso de threads para el manejo de procesos ligeros o hilos en un ambiente de memoria compartida.

1.3. CONCEPTOS BÁSICOS DE PROGRAMACIÓN CONCURRENTE Y PARALELA

En un modelo general de programación, un programa ordinario se forma de un conjunto de instrucciones y datos, escrito en algún lenguaje de programación (comúnmente imperativo) donde las instrucciones son ejecutadas secuencialmente (una tras otra). La ejecución de éste código viene refrendada entonces por su linealidad, es decir, el procesador ejecuta un conjunto de instrucciones de máquina único, que pertenece a un solo proceso. A éste tipo de programas se conoce como programa secuencial (ver Figura 3).

Definimos entonces la programación secuencial como un simple hilo de ejecución o proceso ligero[1] donde un procesador ejecuta un conjunto de instrucciones de manera secuencial.

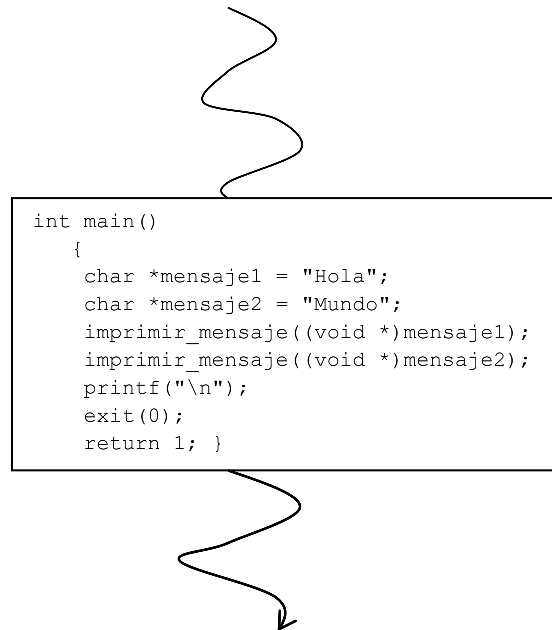


Fig. 3. Un Simple hilo de Ejecución, Proceso Ligero o Programa Secuencial

Por el contrario, en un modelo de Programación Concurrente, existen varios programas que se ejecutan secuencialmente, es decir, existen múltiples hilos de ejecución trabajando en paralelo (ver Figura 4), aparentemente, ya que se trata de un paralelismo abstracto, pues no es necesario utilizar un procesador físico para ejecutar cada proceso. En otras palabras, si observamos en un intervalo de tiempo lo suficientemente amplio el conjunto de instrucciones de máquina ejecutado por el procesador, veremos que hay instrucciones que pertenecen a diferentes hilos de ejecución; pues los distintos procesos comparten el tiempo de ejecución de un único procesador disponible, mediante alguna técnica de planificación[2].

En conclusión diremos que, aunque el programa concurrente sea ejecutado en un único procesador, supondremos que los procesos que lo integran están siendo ejecutados simultáneamente, y no nos preocuparemos por los detalles del paralelismo físico que proporciona nuestra computadora.

[1] En páginas posteriores se hará la distinción entre lo que es un proceso ligero y un proceso pesado.
[2] Esto es así, ya que las instrucciones de los procesos se ejecutan intercalándose unas con otras (paralelismo lógico).

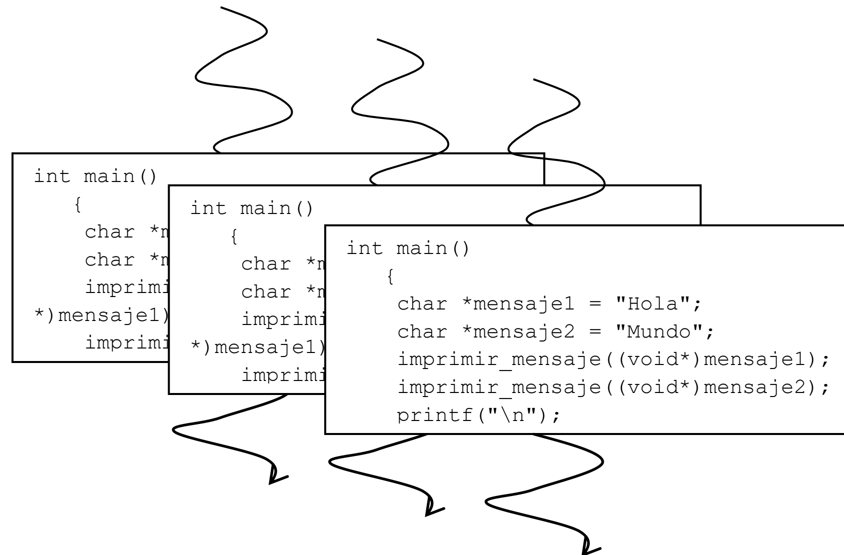


Fig. 4. Programación Concurrente: Múltiples hilos de Ejecución trabajando en "paralelo"

De forma física los procesos pueden:

a) Multiplexar su ejecución en un único procesador (Multiprogramación)
b) Multiplexar su ejecución en un sistema multiprocesador de memoria compartida (Multiproceso)
c) Multiplexar su ejecución en varios procesadores que no comparten memoria (Procesamiento Distribuido)
d) Ejecutarse según un modelo híbrido de los 3 anteriores

Sólo en los casos (b), (c) y (d) se puede hablar de una ejecución paralela verdadera. El término concurrente indica paralelismo potencial.
Una propiedad fundamental de la programación concurrente es el no determinismo: dado un instante de tiempo, no es conocido que va a ocurrir en el instante siguiente. Para la implementación sobre un único procesador, no puede saberse si va a ocurrir o no una interrupción que cause un intercambio del proceso que esta siendo ejecutado. En el caso de un sistema multiprocesador, las velocidades de los procesadores no están sincronizadas, por lo que no puede saberse que procesador va a ser el primero en ejecutar su siguiente instrucción. Comúnmente es deseable que el estado final de un cálculo esté determinado, aunque el cálculo mismo no lo esté. Por ejemplo, es deseable que la suma de dos subexpresiones sea la misma, independientemente de cual de ellas sea evaluada en primer lugar. Esto podría llevarnos a concluir que: Sería

necesario disponer de un modelo abstracto de concurrencia, que permita razonar sobre la corrección o correctitud de programas y sistemas concurrentes.
En cuanto al tipo de interacción entre procesos, se pueden distinguir entre tres tipos de conducta:

a) Procesos Independientes: Aquellos que no se comunican entre sí y por tanto no requieren sincronizarse (ponerse de acuerdo).
b) Procesos Cooperantes: Aquellos que colaboran en la realización de un trabajo común, y para ello, deben comunicarse entre sí y sincronizar sus actividades.
c) Procesos en Competencia: Aquellos que comparten un número finito de recursos de un sistema computacional, por ejemplo, dispositivos periféricos, memoria, capacidad del procesador, etc. Los procesos deben competir "en una carrera" por obtener el uso de los recursos del sistema. Esta competencia requiere que los procesos se comuniquen y/o se sincronicen, aun cuando las labores que realicen sean independientes.

Programación Concurrente

La programación concurrente es el conjunto de notaciones y técnicas utilizadas para describir mediante programas el "paralelismo potencial" de los problemas, así como para resolver los problemas de comunicación y sincronización que se presentan cuando varios procesos que se ejecutan concurrentemente comparten recursos.
Cada problema concurrente presenta un tipo distinto de paralelismo; implementarlo, es una cuestión ligada en principio a la arquitectura de la computadora en cuestión. Para poder trabajar de forma independiente de la arquitectura se requiere utilizar un modelo abstracto de la concurrencia que permita razonar sobre la corrección de los programas que implementen el paralelismo, con independencia de la máquina en que estos programas se ejecuten. Un lenguaje de Programación por ejemplo es una abstracción que sirve al programador para comunicarse con la computadora sin tener que considerar la arquitectura física de ésta o las instrucciones de máquina.
La programación concurrente como abstracción, está diseñada para permitirnos razonar sobre el comportamiento dinámico de los programas y sistemas concurrentes cuando se ejecutan sobre un único procesador, utilizando el paralelismo abstracto ya definido.

Los Procesos

Un proceso es una entidad compuesta de un código ejecutable secuencial, un conjunto de datos y una pila que se utiliza para la transferencia de parámetros, restauración de llamadas recursivas o de interrupciones, etc. Una parte de la pila contiene el entorno del proceso (ver Figura5).

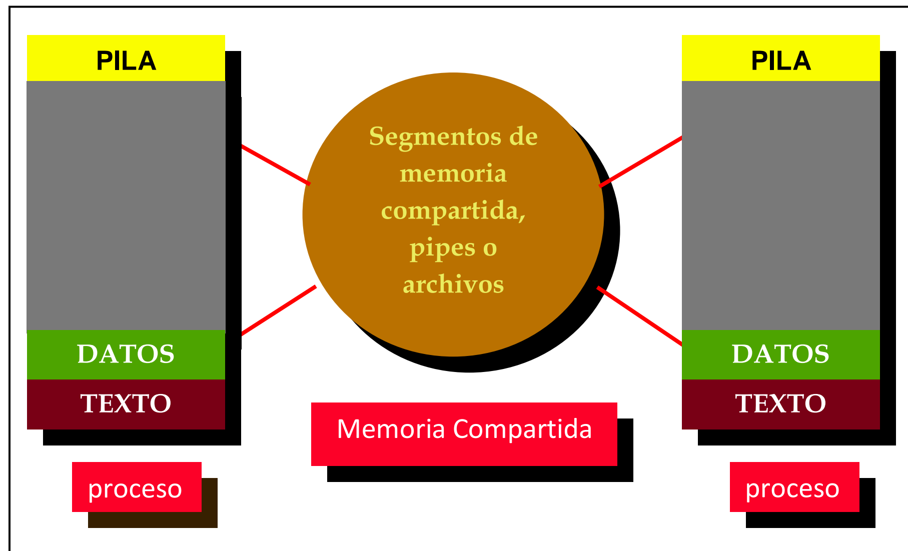


Fig. 5. Modelo básico de un Proceso

Aun cuando el concepto de proceso es "el mismo" en todos los lenguajes de programación concurrente, existen variaciones en cuanto al modelo de concurrencia que adoptan. Estas variantes se dan en aspectos como:

- ***La estructura:***
  - ***Estática:*** Cuando el número de procesos del programa concurrente es fijo y se conoce en tiempo de compilación
  - ***Dinámica:*** Cuando los procesos pueden ser creados en cualquier momento. El número de procesos existentes en el programa concurrente sólo se conoce en tiempo de ejecución.
- ***El Nivel de Paralelismo:***
  - ***Anidado:*** Cuando un proceso puede ser definido dentro de otro proceso.
  - ***Plano:*** Cuando los procesos sólo pueden ser definidos en el nivel más externo del programa.
- ***Relaciones entre Procesos:*** Se pueden crear jerarquías de procesos e interrelaciones entre ellos con el uso de los niveles de paralelismo anidados.
  - ***Relación Padre/Hijo:*** Un proceso (el padre) es el responsable de la creación de otro, el hijo. El padre ha de esperar mientras el hijo está siendo creado e inicializado.

- ***Relación Guardian/Dependiente:*** El proceso guardián no puede terminar hasta que todos los procesos dependientes hayan terminado[3].

- ***Granularidad:*** Se puede hacer la división de paralelismo de grano fino y paralelismo de grano grueso. Un programa concurrente de grano grueso contiene relativamente pocos procesos, cada uno con una labor significativa que realizar. Los programas de grano fino contienen un gran número de procesos, algunos de los cuales pueden incluir una única acción.

Estados y Operaciones de un Proceso

Durante el tiempo de existencia de un proceso, éste se puede encontrar en uno de varios estados posibles. El cambio de un estado a otro depende de ciertas operaciones (ver Figura 6):

- ***Nuevo:*** El proceso ha sido creado por un usuario al lanzar un programa concurrente a ejecución.
- ***En Ejecución:*** Las instrucciones de máquina que conforman el código fuente del proceso están siendo ejecutadas por el procesador.
- ***Bloqueado:*** El proceso está esperando a que ocurra algún tipo de evento, por ejemplo, la realización de una operación de entrada/salida.
- ***Listo:*** El proceso está en espera de que sus instrucciones sean ejecutadas por el procesador cuando el planificador del sistema operativo le dé el turno para ello.
- ***Finalizado:*** El proceso ha terminado y todos los recursos que ha utilizado para ejecutarse son liberados.

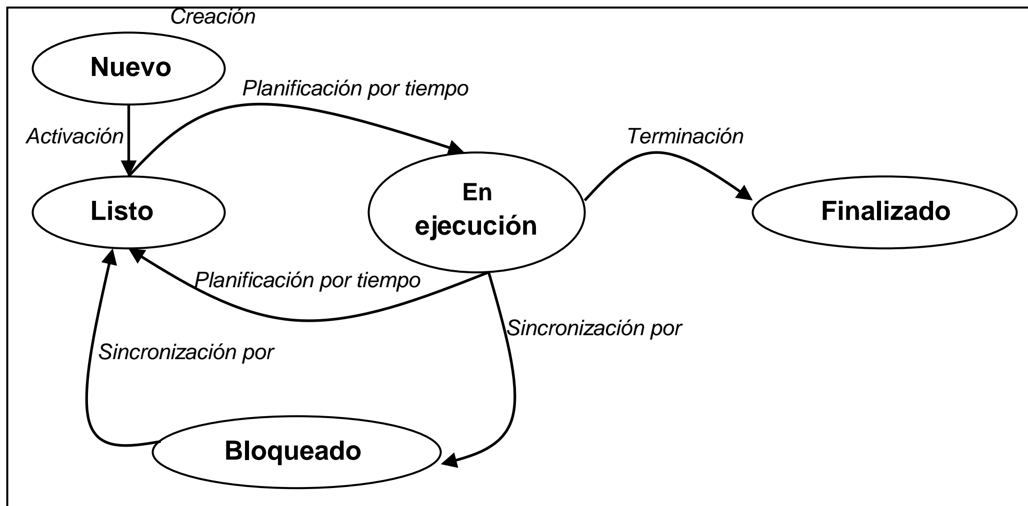


Fig. 6. Diagrama de Transición de Estados de un Proceso a través de operaciones

[3] En lenguajes que permiten estructuras dinámicas (como C o Ada), el padre y el guardián pueden no ser el mismo.

La acción que hace que un proceso cambie de un estado a otro es una operación. Los procesos llevan asociadas diversas operaciones: creación y activación, terminación, sincronización y planificación (scheduling).

1. ***Creación y Activación:*** Se distingue entre creación y activación porque en algunos lenguajes y sistemas, los procesos pueden ser declarados (creados), pero no comienzan (se activan) hasta que se alcanza un punto determinado en el programa que los crean. En algunos casos, la activación es implícita cuando se alcanza ese punto.
2. ***Terminación:*** La forma en que un proceso termina difiere entre los sistemas. Un proceso puede terminar implícitamente, cuando acaba de ejecutar sus tareas y no tiene procesos dependientes, o explícitamente, mediante la ejecución de una operación.
3. ***Planificación o Scheduling:*** La planificación es el método por el que los procesos son asignados a los procesadores disponibles, es decir; es el medio por el que un proceso pasa del estado *listo* al estado *en ejecución*.
4. ***Sincronización:*** Las operaciones de sincronización entre procesos tienen que ver con técnicas de comunicación a través de mecanismos como la exclusión mutua, candados, variables de condición, semáforos o monitores; mecanismos de los que más adelante hablaremos en apartados especiales para ello.

Formas de crear y manejar procesos:

1. ***Llamadas al Sistema Operativo:*** los que están a favor de esta forma de hacer programación concurrente dicen que es la manera por la que se consigue mayor eficiencia en la ejecución de los programas. Los que están en contra dicen que los programas que utilizan servicios del sistema operativo son más complejos, menos obvios y con un alto costo para ser adaptados a otro sistema operativo.
2. ***Lenguajes de Programación que soportan Programación Concurrente con Independencia del Sistema Operativo:*** Ejemplos de lenguajes de programación que incorporan estructuras de soporte de concurrencia son: Ada, Occam, Pascal-Concurrente, C-Concurrente, Java, etc. Los que están a favor de esta manera de hacer concurrencia dicen que ésta se expresa de un modo claramente visible para el programador, los programas son más fáciles de probar y mantener, se mejora el poder de abstracción, la modelización del mundo físico es más clara y natural, los programas son menos dependientes de la máquina.
3. ***Bibliotecas para creación y operaciones con procesos, independientes del S.O.:*** Según éste método, el lenguaje no conlleva estructuras sintácticas especiales. En su lugar, una biblioteca soporta la concurrencia mediante la definición de tipos y operaciones que pueden ser invocados por una aplicación. La interfaz entre el

programador y dichos módulos es independiente de la máquina y del sistema. Un ejemplo de esto es la forma de concurrencia con el uso de la biblioteca ***Pthreads*** del lenguaje C.

Multiprogramación, Multitarea y Procesos

La multitarea se encuentra en el núcleo de casi cualquier sistema operativo moderno y surge debido a la necesidad de aprovechar los tiempos muertos de la computadora dejados por u proceso cuando efectúa una operación de E/S; la solución para llenarlos es dejar que otro proceso comience a ejecutarse. De ahí que la multiprogramación como ya se ha dicho en líneas anteriores se defina como la ejecución concurrente de varios procesos independientes sobre un solo procesador.

Para lograr la multiprogramación se utiliza un programa planificador (scheduler) que ejecutado por el S.O. y quizás considerando algún esquema de prioridades, determina a qué proceso le corresponde acceder al procesador.

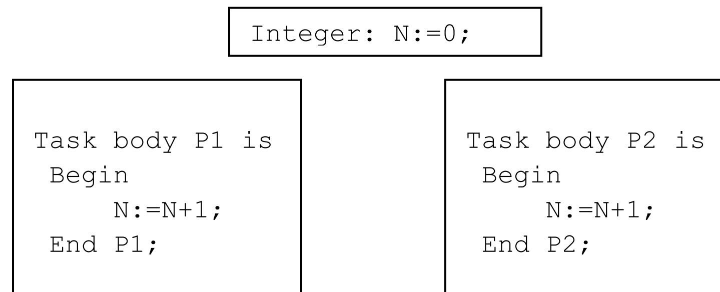
La generalización de la multiprogramación nos lleva al concepto de paralelismo, donde se intenta resolver un problema descomponiéndolo en varios procesos concurrentes.
Interfoliación: La interfoliación aparece en la concurrencia abstracta y se define como el entrecruzamiento en un único código secuencial real de los códigos concurrentes de varios procesos, a nivel de instrucciones atómicas. Aparece entonces el no-determinismo de la aplicación, que como ya se citó, es una de las características fundamentales de la programación concurrente y es el que dificulta el análisis de la corrección semántica del programa. Por ejemplo, suponga que se lanzan dos procesos concurrentes que modifican una variable compartida x:

```
Proceso 1:
   Load(x);
   Add(x,1);
   Store(x);
```

```
Proceso 2:
   Load(x);
   Add(x,1);
   Store(x);
```

Al lanzar ambos procesos de forma concurrente, es posible que se den dos secuencias de interfoliación distintas que hacen que la variable x pueda tomar distintos valores (uno o dos). En consecuencia, dos ejecuciones distintas de un mismo programa concurrente llevarán asociadas diferentes secuencias de interfoliación que darán lugar a resultados distintos.

Parece entonces que la depuración de un programa concurrente se torna inviable, siendo inútil el uso de un trazador, pues cada ejecución del programa origina una interfoliación diferente. Esto origina incertidumbre sobre el resultado final y no consideraremos correcto a un programa concurrente a menos que lo sea bajo todas las secuencias posibles de interfoliación[4]. Si consideramos una posible versión de alto nivel del pseudocódigo anterior, tendríamos:



```
Integer: N:=0;

Task body P1 is
 Begin
     N:=N+1;
 End P1;

Task body P2 is
 Begin
     N:=N+1;
 End P2;
```

Se pueden dar dos circunstancias:

- Que el compilador traduzca el programa en una simple instrucción ***INC***, en cuyo caso el entrelazado no afecta el resultado.
- Que el compilador traduzca el programa en código de máquina, en cuyo caso utilizaría registros del procesador y los entrelazados podrían dar lugar a distintas salidas, algunas erróneas como la que se muestra en la siguiente tabla:

PROCESO	INSTRUCCION	N	REGISTRO 1	REGISTRO 2
Inicial		0		
P1	Load(x)	0	0	
P2	Load(x)	0	0	0
P1	Add(x,1)	0	1	0
P2	Add(x,1)	0	1	1
P1	Store(x)	1	1	1
P2	Store(x)	1	1	1

En la tabla se muestra una posible interfoliación que da un error, pues cualquier persona esperaría observar un valor de x=2.

[4] La interfoliación entre procesos concurrente se da a nivel de instrucciones atómicas.

Hilos o Threads

Un método para lograr el paralelismo consiste en hacer que varios procesos cooperen y se sincronicen mediante la compartición de memoria. Una alternativa para hacerlo viable es la de emplear múltiples threads (hilos) de ejecución en un solo espacio de direcciones. Un thread es una secuencia de instrucciones ejecutada dentro de un programa, en otras palabras, cuando se ejecuta un programa, el CPU utiliza el contador de programa del proceso para determinar qué instrucción debe ejecutarse a continuación.

El flujo de instrucciones resultante se denomina hilo de ejecución del programa y se puede ver como el flujo de control para el proceso, representado por una secuencia de direcciones de instrucciones que el contador de programa va indicando durante la ejecución [BUT97]. Desde el punto de vista del programa, la secuencia de instrucciones de un hilo de ejecución es un flujo ininterrumpido de direcciones; en cambio, desde el punto de vista del procesador, los hilos de ejecución de diferentes procesos están entremezclados y el punto en el que la ejecución cambia de un proceso a otro es denominado conmutación de contextos. Además, una extensión natural del modelo de proceso concurrente es el permitir la ejecución de varios threads dentro del mismo proceso, que recibe el nombre de multithreading, (ver figura 7), que constituye un mecanismo eficiente para controlar hilos de ejecución que comparten tanto código como datos en una aplicación paralela, con lo cual se evitan las conmutaciones de contexto. Esta estrategia también mejora el rendimiento, pues, en una máquina multiprocesador, a los procesadores se les puede asignar múltiples threads de tal forma que estas consiguen ser ejecutadas simultáneamente.

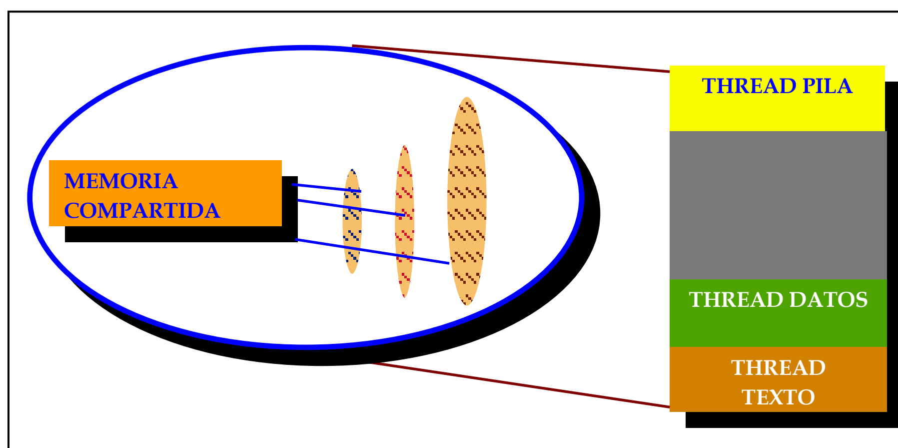


Fig. 7. Threads dentro de un proceso (todos los threads son parte de un proceso)

Cada hilo de ejecución se asocia con un thread, es decir, con un tipo de datos abstracto que representa el flujo de control dentro de un proceso. Cada thread tiene su propia pila de ejecución, valor de contador de programa, conjunto de registros y estado (ver Figura 8).

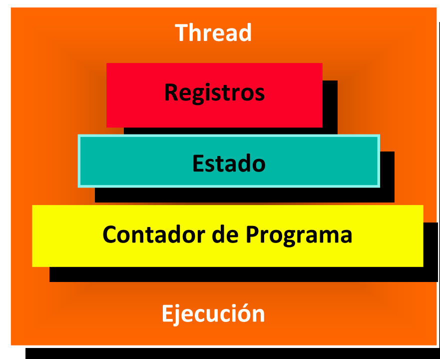


Figura 8. Modelo genérico de un Thread

Al declarar muchos threads dentro de los confines de un solo proceso, el programador puede lograr paralelismo con un bajo costo[5;] sin embargo, los threads también presentan ciertas complicaciones en cuanto a la necesidad de sincronización. Los Threads son frecuentemente llamados procesos ligeros y puede decirse que son "primos" de los procesos UNIX. En UNIX cualquier proceso ha de poseer los siguientes elementos constitutivos:

1. Un espacio de direcciones (pila, datos y segmento de código).
2. Una tabla de descriptores de archivos (archivos abiertos).
3. Un hilo o programa de ejecución.

Características de los Threads

- Los *threads* son más pequeños comparados con los procesos.
- La creación de un *thread* es relativamente menos costosa.
- Los *threads* comparten los recursos mientras que los procesos requieren su propio conjunto de recursos.
- Los *threads* ocupan menos memoria (es decir, son más económicos respecto del gasto de recursos computacionales en un sistema).

[5] La figura 5 muestra el modelo básico de un proceso con estas características.

- Los *threads* proporcionan a los programadores la posibilidad de escribir aplicaciones concurrentes que se pueden ejecutar tanto en sistemas monoprocesador, como en sistemas multiprocesador de forma transparente.
- Los *threads* pueden incrementar el rendimiento en entornos monoprocesador.

Con Threads o sin Threads

Los threads no necesariamente proporcionan la mejor solución a cualquier problema de programación. No siempre son fáciles de usar y no siempre proporcionan el mejor rendimiento. Existen muchos problemas que son inherentemente no concurrentes, por lo que, añadirles threads puede incrementar innecesariamente el código aumentado, el tiempo de programación y también complicar la programación de las aplicaciones. Si cualquier paso o secuencia en un programa depende directamente del resultado del paso previo, entonces el usar threads probablemente no ayudará en nada, ya que cada thread tendría que esperar a otro thread para poder ser completado. Los candidatos más obvios para codificar threads son aplicaciones que contemplen:

- realización de gran cantidad de cálculos que puedan ser paralelizables (o descompuestos) en múltiples threads, en donde dichos cálculos puedan intentar ejecutarse en multiprocesadores de hardware,
- bien la necesidad de realizar muchas operaciones de E/S que puedan ser superpuestas (es decir, donde muchos threads puedan esperar a diferentes peticiones de E/S al mismo tiempo). Los sistemas servidores o las aplicaciones distribuidas son buenos candidatos para ello.

Programación con Threads

Un paquete o biblioteca de threads junto con algún lenguaje de programación específico permite escribir programas con varios puntos simultáneos de ejecución sincronizados a través de memoria compartida. Sin embargo, la programación con threads introduce nuevas dificultades, ya que la programación concurrente y paralela acomete problemas que no se suscitan en la programación secuencial, así como también se utilizan técnicas diferentes para resolverlos. Existen problemas simples, por ejemplo, el interbloqueo entre procesos que se adaptan mejor a ser resueltos con un ambiente de ejecución con threads, pero para otros problemas el rendimiento de las aplicaciones resulta ser penalizado.

Un thread es un concepto sencillo: un simple flujo de control secuencial [KLE96]. El programador no necesita aprender nada nuevo para emplear un único thread en el desarrollo de un programa. Cuando se tienen múltiples threads en un programa, significa que en cualquier instante el programa tiene múltiples puntos de ejecución, uno en cada uno de sus threads; por lo que el programador ha de decidir cuándo y dónde crear los múltiples threads. Las operaciones de creación vienen proporcionadas mediante la correcta utilización de un paquete de librería o de un sistema con threads, pero las directrices del diseño no suelen ser evidentes para un programador no acostumbrado al desarrollo de aplicaciones paralelas. Por otro lado, en un lenguaje de alto nivel, las variables globales son compartidas por todos los threads del programa, es decir, leen y escriben en las mismas posiciones de memoria; sin embargo, el programador es el responsable de emplear los mecanismos de sincronización del paquete de threads para garantizar que la memoria compartida se acceda de manera correcta, lo cual hace más propensas a las aplicaciones a contener errores de difícil depuración que afectan a la seguridad de los programas.

Las facilidades proporcionadas por un paquete del tipo anterior son conocidas como primitivas ligeras, lo que significa que las primitivas de:

- Creación,
- Mantenimiento,
- Sincronización y
- Destrucción,

Debieran ser lo suficientemente accesibles para limitar el esfuerzo necesario para la correcta utilización que cubra las necesidades de concurrencia del programador[6].

Los Threads en la Concurrencia

- ***En el uso de un sistema multiprocesador:*** Los *threads* son una herramienta atractiva para permitir a un programa sacar provecho de las facilidades para mejorar el *speedup o rendimiento* de las aplicaciones paralelas que presenta este tipo de hardware. Empleando un paquete de *threads*, el programador puede utilizar los procesadores de forma económica en cuanto al aprovechamiento de la capacidad computacional de estos. Parece funcionar bien en sistemas que van de 10 a 1000 procesadores.
- ***En la gestión de dispositivos de entrada/salida:*** Estos dispositivos pueden ser programados fácilmente mediante *threads*, de tal forma que las peticiones a un servicio sean secuenciales y el *thread* que realiza la petición quede suspendido hasta que la solicitud sea completada; mientras, el programa principal puede realizar más trabajo con otros *threads* solapando la ejecución de varios hilos.

[6] Esto normalmente no es así, ya que las librerías de threads presentan muchas funciones, con un gran número de parámetros algunas de ellas, que no hacen intuitiva, ni mucho menos, su utilización.

- ***Con los usuarios como fuente de concurrencia:*** A veces un usuario necesita realizar dos o tres tareas simultáneamente, la utilización de los *Threads* en las aplicaciones son una buena forma de atender esta necesidad.
- ***En la constitución de un sistema distribuido:*** Servidores de red compartidos, donde el servidor se encarga de recibir peticiones de múltiples clientes. El uso de múltiples *threads* permite al servidor gestionar las solicitudes de los clientes en paralelo, en vez de procesarlas en serie, lo que hubiera significado la creación de un proceso de servicio para cada cliente con un enorme desperdicio de la capacidad de procesamiento.
- ***En la reducción de la latencia[7] de las operaciones de un programa:*** Emplear *threads* para diferir el trabajo es una técnica que proporciona muy buenos resultados. La reducción de la latencia puede mejorar los tiempos de respuesta de un programa.

Finalmente, uno de los problemas de utilizar múltiples *threads* de ejecución es que hasta hace poco no existía un estándar. Algunos modelos populares de *threads* son:

- Mach C threads, CMU
- Sun OS LWP threads, Sun Microsystems
- PARAS CORE threads, C-DAC
- Java-Threads, Sun Microsystems
- Chorus threads, Paris
- OS/2 threads, IBM
- Windows NT/95 threads, Microsoft
- POSIX, ISO/IEEE standard

La extensión POSIX.1c se aprobó en Junio de 1995. El estándar POSIX-Thread significa técnicamente el API-Thread especificado por el estándar formal internacional POSIX[8] 1003.1c-1995. Con la adopción de un estándar POSIX para los threads, las aplicaciones comerciales actuales se han ido construyendo con este API y seguramente en un futuro no muy lejano dicha estandarización será adoptada por todos.

[7] La latencia es el tiempo empleado entre la llamada a un procedimiento y la finalización de éste.
[8] POSIX significa *Portable Operating System Interface*.