Introducción a la computación paralela

Breve historia de la computación paralela

Ilustración 1 Procesador Intel 4004 un procesador de 4 bits

La computación paralela no es un concepto relativamente nuevo. Cuando Intel desarrolló su primer procesador comercial el Intel 4004 en 1971, un procesador de 4 bits que era capaz de procesar hasta cuatro bits de datos simultáneamente en operaciones como la suma y en operaciones lógicas, siempre en grupos de 4 bits. Efectivamente, el procesador tenía la capacidad de procesar datos de cuatro bits a la vez por instrucción más no podía ejecutar más de una instrucción al mismo tiempo. A ésta capacidad de procesamiento paralelo, se le conoce como procesamiento paralelo a nivel de bits. El nivel de paralelismo a nivel de bits fue creciendo con el tiempo a 8 bits (i8008), luego a 16 bits (i8086), 32 (i80386) pero ejecutando solo una instrucción a la vez. A éste tipo de procesadores se les conoce como SISD (Single Instruction-Single Data).

Ilustración 2 Cray-1 En el Laboratorio Nacional de Los Alamos California EU.

Ya en 1976 el supercomputador Cray-1 de la empresa Cray Research Company ofrecía una arquitectura de procesador vectorial de elementos de 64 bits. El procesador vectorial era capaz de realizar una instrucción sobre dos vectores de datos. Estos procesadores se les conoce como SIMD (Single Instruction-Multiple Data), en la que se realiza la misma operación aritmético-lógica sobre un conjunto de datos al mismo tiempo. El paralelismo es cada vez más evidente. Contaba con 16MB de RAM y con un reloj de 80MHz a un costo de $5 MDD a $8MDD .

Personalidades importantes como Gene Amdhal en la década de 1960, propone la ley de Amdhal en la que se establece la que velocidad potencial del computador paralelo está limitado por la porción del algoritmo que no puede paralelizarse. La ley de Amdhal nos indica perfectamente el reto de la programación paralela para obtener los beneficios de los computadores modernos.

En la década de 1980 los fabricantes de semiconductores mejoraron sus procesos de producción y aumentaron la cantidad de transistores dispuestos por unidad de área y los costos fueron disminuyendo, permitiendo en 1981 a IBM que los computadores personales IBM PC llegaran a las pequeñas y medianas empresas así como el hogar.

Sin embargo no es sino hasta la década de 1990 cuando los procesadores MIMD (Multiple Instruction – Multiple Data) capaces de ejecutar más de una instrucción sobre múltiples datos

estuvieron disponibles. A estos procesadores también se les conoce como superescalares ya que el procesador cuenta con más de una unidad de ejecución y permitiendo la ejecución de dos o más instrucciones simultáneamente, siempre y cuando el estado entre estas ejecuciones no tengan dependencia entre sí. Ejemplos de computadores con éstas características de cómputo MIMD fueron aquellos que equiparon el procesador Intel Pentium MMX, Pentium II y hasta el más moderno Intel Core i7, éstos procesadores cuentan varias unidades de ejecución SIMD posibilitando la ejecición de dos o más instrucciones al mismo tiempo. Esto impulsó el desarrollo multimedia y del entretenimiento de la época ya que el procesamiento de audio, video requieren gran poder de cómputo durante la decodificación y codificación.

Ilustración 3 CPU Intel Core i7

Luego llegan los procesadores *multinúcleo, es decir, en un solo* encapsulado dos o más CPU's completos e interconectados compartiendo la memoria y el acceso a los periféricos. Ofreciendo así más paralelismo al doble o por el triple dependiendo del número de éstos núcleos integrados en lo que hoy conocemos simplemente como CPU.

Ilustración 4 GPU Radeon de AMD

Actualmente el GPU o Unidad de Procesamiento Gráfico es un coprocesador dedicado al cómputo de tareas gráficas en la que gran cantidad de procesadores de tipo SIMD más simples que un CPU, son encapsulados en un solo chip. Esta reciente arquitectura nos ha permitido procesar grandes cantidades de información mediante operaciones simples y en paralelo. Por lo generar, a un determinado grupo de procesadores se les asigna una sencilla tarea, son más lentos si se les considera individualmente, pero en grupo, el poder de cómputo se multiplica. Cada grupo de unidades SIMD ejecuta el mismo programa, por lo que el procesamiento es paralelo y de forma sincrónica.

Es común encontrar en el mercado moderno CPU's con hasta 8 núcleos de procesamiento y GPU's con hasta 2,048 núcleos.

Multitarea y paralelismo

Las palabras multitarea y paralelismo suelen confundirse o a veces se utilizan intercambiablemente en textos que hablan de estos temas. Sin embargo, distinguir los significados de éstas palabras nos ayudará a descubrir propiedades interesantes acerca de los sistemas de procesamiento paralelo.

La multitarea en particular, es la capacidad de un sistema de computación de realizar varias tareas aparentemente, y aclaro aparentemente, casi al mismo tiempo. Es muy común encontrar la palabra multitarea cuando se habla de sistemas operativos, y efectivamente son éstos sistemas los que buscan compartir los recursos de computación del procesador principal ejecutando tareas a manera de turnos en forma progresiva y justa.

Para aclarar lo anterior, supongamos que tenemos un computador con un solo procesador. ¿Se podrá realizar multitarea en éste procesador? La respuesta es “Si”. ¿Cómo? Es la segunda pregunta. Bueno, en realidad lo que hace el computador es ejecutar un programa administrador (el sistema operativo, claro), capaz de distribuir los recursos de ejecución entre varias tareas a su cargo. También supongamos que actualmente se está ejecutando una tarea la cual goza de los recursos de ejecución del único procesador principal. Luego tras un determinado tiempo en la que la tarea es ejecutada y se sigan cumpliendo ciertas condiciones, el sistema operativo retira los recursos de ejecución y almacena su contexto de ejecución. El contexto de ejecución es el estado actual del modelo de programación (registros, pila y contador de programa) justo en el instante que se suspendió la tarea. Guardar el contexto de ejecución es importante para que posteriormente se reasuma la ejecución de las tareas en el mismo punto donde fueron interrumpidas. Tras el salvado del contexto, el sistema operativo restaura el contexto de ejecución de otra tarea y asigna los recursos de ejecución a esa otra tarea. Esto se repite indefinidamente, de tal manera que todas las tareas puedan ejecutarse de manera progresiva. Debido a que éste intercambio de contextos de ejecución entre tareas se realiza muy rápido por parte del computador, ante el ser humano las tareas parecieran ejecutarse “al mismo tiempo”. Un simple truco de percepción, que resulta muy útil en aplicaciones prácticas cuando solo se cuenta con un solo procesador y con la necesidad de ejecutar varias tareas “casi simultáneamente”.

Puede observarse que el procesador, o ejecutaba tareas o ejecutaba al sistema operativo, pero jamás al mismo tiempo. El sistema operativo puede considerarse así como una tarea administradora de otras tareas.

La multitarea, como podrá ahora comprender, no implica necesariamente paralelismo de tareas. Pero es una fascinante característica que no podemos pasar por alto. Porque la vamos necesitar, incluso, cuando tengamos disponible la capacidad de procesamiento paralelo.

Ahora bien sigamos definiendo el paralelismo. El paralelismo es una característica del sistema computacional de realizar dos o más tareas simultáneamente, y aclaro simultáneamente, al mismo tiempo, concurrentemente.

Al principio comentamos acerca del paralelismo a nivel de bits, lo cual sigue siendo verdadero para nuestro ejemplo hipotético anterior en donde se mostró el concepto de multitarea, pero ahora nos vamos a referir al paralelismo a nivel de tareas, el cual es de nuestro principal interés de estudio.

Para que ocurra el paralelismo o concurrencia, es necesario que el computador cuente con un procesador capaz de ejecutar dos o más instrucciones de una misma tarea simultáneamente, logrando así el *paralelismo a nivel de instrucción*. También el computador puede contar con varios núcleos de procesamiento lo que permite el *paralelismo a nivel de tarea*. Es también un hecho que existen procesadores que pueden ejecutar dos o más instrucciones de distintas tareas al mismo tiempo de un mismo núcleo, lo que permite aprovechar los recursos de ejecución disponibles.

Ahora que entendemos bien la diferencia entre multitarea y procesamiento paralelo recomiendo que reflexiones con el siguiente conjunto de ejercicios que reforzarán tu aprendizaje de éstos interesantes temas.

Ejercicios, contesta a las siguientes preguntas y realiza las actividades indicadas.

¿Es cierto que en lo general el paralelismo implica multitarea?

¿Por qué en lo general la multitarea no implica paralelismo?

¿Por qué crees que la programación paralela representa un reto para el programador?

¿Por qué el sistema operativo cumple un rol importante como administrador de los recursos de ejecución?

¿Por qué el GPU tiene más procesadores o núcleos que un CPU? ¿A caso el GPU puede hacer lo mismo que el CPU?

¿Por qué usamos CPU’s como procesadores principales y GPU’s como coprocesadores?

Identifica las características de procesamiento paralelo y de multitarea de los siguientes sistemas operativos: PC-DOS de IBM, MS-DOS de Microsoft, Linux 2.6.x , Android y Windows 2000/XP/7/8

Computación Vectorial SIMD

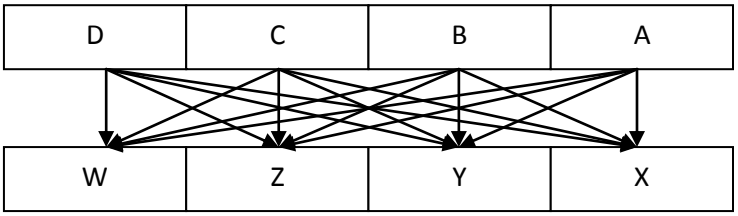
Antes de los computadores con capacidad SIMD, estuvieron disponibles los procesados SISD, (Single Instruction, Single Data) en donde las instrucciones operan con varios operandos escalares.

La computación vectorial SIMD es otra forma de *paralelismo a nivel de datos*, de manera semejante al *paralelismo a nivel de bits* que ya se lograba con la computación SISD. Una operación o instrucción SIMD puede comprenderse fácilmente si trabajamos con operandos vectoriales (de allí su nombre) en vez de simples escalares. Cada instrucción SIMD, está diseñada de tal manera que opere con uno o más vectores. Por ejemplo, supongamos que existe una instrucción SIMD llamada 'Sumar' y supongamos vectores de la forma [X, Y, Z, W], entonces la suma se puede realizar entre dos vectores así:

[a, b, c, d] + [e, f, g, h] = [a+e, b+f, c+g, d+h]

Los procesadores actuales pueden realizar la operación anterior, realizando cuatro sumas paralelas mediante una sola instrucción SIMD.

Otras de las instrucciones SIMD más utilizadas son los intercambios, que permiten copiar los elementos de un vector origen en un vector destino así:



Las instrucciones de intercambio son importantes ya que es una forma especial de movimiento de datos en paralelo. Las instrucciones de intercambio también requieren un parámetro que indique cómo se copiarán los datos del origen al destino. Cuando los vectores son de cuatro elementos, existen 256 movimientos posibles, es decir 4^4, por lo que el parámetro de intercambio es de 8 bits. Si nuestro operando a permutar es un vector de tres elementos, entonces existen $3^3 = 27$ movimientos posibles, por lo que se requieren solo 5 bits para especificar el parámetro de intercambio. En los computadores modernos es común encontrar instrucciones de intercambio con vectores tetradimensionales. Es claro, que se puede hacer intercambios con vectores tridimensionales si ignoramos un cuarto elemento cualquiera de un vector tetradimensional.

Las instrucciones SIMD pueden clasificarse en dos categorías importantes: *Instrucciones SIMD Horizontales* e *Instrucciones SIMD Verticales.*

Una instrucción SIMD vertical realiza operaciones elemento a elemento entre dos vectores, como por ejemplo el producto elemento a elemento entre dos vectores así:

ProductoVertical([a, b, c, d] , [e, f, g, h]) = [a*e, b*f, c*g, d*h]

Ejemplos de instrucciones verticales serían: la suma elemento a elemento entre dos vectores, la diferencia de dos vectores, la suma de dos vectores, división elemento a elemento entre dos vectores, etc.

Una instrucción SIMD horizontal, opera con los elementos de un vector para producir el resutado Por ejemplo, la suma de los elementos de un vector es una operación horizontal como se muestra a continuación:

SumaHorizontal([a, b, c, d]) = a + b + c + d

Otros ejemplos de instrucciones SIMD horizontales serían: el máximo elemento de un vector, el mínimo elemento de un vector, la media aritmética de los elementos de un vector, la moda de los elementos de un vector, la magnitud de un vector.

El siguiente ejemplo de programación muestra un simple ejemplo en C++ con ensamblador en línea que permite mostrar el uso de instrucciones SIMD. C/C++ son lenguajes SISD, donde el único paralelismo disponible sintáctica y semánticamente es a nivel de bits. Recurrimos al lenguaje ensamblador para utilizar las instrucciones SIMD de un procesador de arquitectura intel x86. El ejemplo muestra la versión secuencial SISD y luego se muestra la versión paralela SIMD para el cómputo del producto punto de dos vectores tetradimensionales.

Nota: El curso no requiere conocimiento de ensamblador, sólo se muestra este ejemplo para que conozca el paralelismo a nivel de datos mediante instrucciones SIMD.

```
/*
	Programa secuencial de ejemplo de procesamiento paralelo
	de datos mediante instrucciones SIMD.

	Producto Punto de dos vectores tetradimensionales
*/
#include "stdafx.h"

union VECTOR4D
{
	struct
	{
		float x,y,z,w;
	};
	float v[4];
};
int _tmain(int argc, _TCHAR* argv[])
{
	VECTOR4D A={0,1,2,3};  //Un primer vector
	VECTOR4D B={4,5,6,7};  //Un segundo vector
	float    s=0.0f;           //El resultado
	//Método secuencial en C/C++
	//Para obtener el resultado
	for(int i=0;i<4;i++)
	{
		s+=A.v[i]*B.v[i];
	}
	printf("%s:%f","El producto punto es mediante SISD:",s);

	//Versión con instrucciones SIMD
	s=0.0f;
	__asm
	{
		movups xmm0,A  //Carga el vector A en xmm0
		movups xmm1,B  //Carga el vector B en xmm1
		mulps  xmm0,xmm1 //Realiza el producto vertical entre A y B: xmm0=xmm0*xmm1
		haddps xmm0,xmm0 //Realiza suma horizontal [d+c,b+a,w+z,y+x] = [w,z,y,x] horz+ [d,c,b,a]
		haddps xmm0,xmm0 //Realiza suma horizontal "
```

```
                movd    s,xmm0    //Almacena el resultado
        }
        printf("%s:%f","El producto punto es mediante SIMD:",s);
        return 0;
}
```

Para verificar la diferencia SISD/SIMD entre estos códigos, a continuación muestro el desensamblado del programa anterior, para hacer una comparativa más detallada: El bloque de código rojo pertenece a la versión SISD, y el bloque azul pertenece la versión SIMD.

```
00191002  in          al,dx
00191003  and         esp,0FFFFFFC0h
00191006  sub         esp,3Ch
00191009  mov         eax,dword ptr [___security_cookie (193000h)]
0019100E  xor         eax,esp
00191010  mov         dword ptr [esp+38h],eax
          VECTOR4D A={0,1,2,3};  //Un primer vector
00191014  fldz
00191016  push        esi
00191017  fstp        dword ptr [esp+2Ch]
          VECTOR4D B={4,5,6,7};  //Un segundo vector
          float    s=0.0f;            //El resultado
          //Método secuencial en C/C++
          //Para obtener el resultado
          for(int i=0;i<4;i++)
          {
                    s+=A.v[i]*B.v[i];
          }
          printf("%s:%f","El producto punto es mediante SISD:",s);
0019101B  mov         esi,dword ptr [__imp__printf (1920A0h)]
00191021  fld1
00191023  sub         esp,8
00191026  fstp        dword ptr [esp+38h]
0019102A  fld         dword ptr [__real@40000000 (19217Ch)]
00191030  fstp        dword ptr [esp+3Ch]
00191034  fld         dword ptr [__real@40400000 (192178h)]
0019103A  fstp        dword ptr [esp+40h]
0019103E  fld         dword ptr [__real@40800000 (192174h)]
00191044  fstp        dword ptr [esp+24h]
00191048  fld         dword ptr [__real@40a00000 (192170h)]
0019104E  fstp        dword ptr [esp+28h]
00191052  fld         dword ptr [__real@40c00000 (19216Ch)]
00191058  fstp        dword ptr [esp+2Ch]
0019105C  fld         dword ptr [__real@40e00000 (192168h)]
00191062  fstp        dword ptr [esp+30h]
00191066  fld         qword ptr [__real@4010000000000000 (192160h)]
0019106C  fldz
0019106E  fmul        st(1),st
00191070  faddp       st(1),st
00191072  fstp        dword ptr [esp+20h]
00191076  fld         dword ptr [esp+20h]
0019107A  fadd        qword ptr [__real@4014000000000000 (192158h)]
00191080  fstp        dword ptr [esp+20h]
00191084  fld         dword ptr [esp+20h]
00191088  fadd        qword ptr [__real@4028000000000000 (192150h)]
0019108E  fstp        dword ptr [esp+20h]
00191092  fld         dword ptr [esp+20h]
00191096  fadd        qword ptr [__real@4035000000000000 (192148h)]
0019109C  fstp        dword ptr [esp+20h]
001910A0  fld         dword ptr [esp+20h]
001910A4  fstp        qword ptr [esp]
001910A7  push        offset string "El producto punto es mediante SI"... (1920F4h)
001910AC  push        offset string "%s:%f" (192118h)
001910B1  call        esi

          //Versión con instrucciones SIMD
          s=0.0f;
001910B3  fldz
001910B5  fstp        dword ptr [esp+28h]
001910B9  add         esp,10h
          __asm
          {
                    movups xmm0,A  //Carga el vector contenido de A en xmm0
001910BC  movups       xmm0,xmmword ptr [esp+2Ch]
                    movups xmm1,B  //Carga el vector contenido de B en xmm1
001910C1  movups       xmm1,xmmword ptr [esp+1Ch]
                    mulps  xmm0,xmm1 //Realiza el producto elemento a elemento entre A y B: xmm0=xmm0*xmm1
001910C6  mulps        xmm0,xmm1
                    haddps xmm0,xmm0 //Realiza suma horizontal [w,z,y,x] horz+ [d,c,b,a] = [d+c,b+a,w+z,y+x]
001910C9  haddps       xmm0,xmm0
                    haddps xmm0,xmm0 //Realiza suma horizontal "
001910CD  haddps       xmm0,xmm0
                    movd   s,xmm0    //Almacena el resultado
001910D1  movd         dword ptr [esp+18h],xmm0
```

```
          }
          printf("%s:%f","El producto punto es mediante SIMD:",s);
001910D7  fld         dword ptr [esp+18h]
001910DB  sub         esp,8
001910DE  fstp        qword ptr [esp]
001910E1  push        offset string "El producto punto es mediante SI"... (192120h)
001910E6  push        offset string "%s:%f" (192118h)
001910EB  call        esi
          return 0;
}
001910ED  mov         ecx,dword ptr [esp+4Ch]
001910F1  add         esp,10h
001910F4  pop         esi
001910F5  xor         ecx,esp
001910F7  xor         eax,eax
001910F9  call        __security_check_cookie (191102h)
001910FE  mov         esp,ebp
00191100  pop         ebp
00191101  ret
```

Analizando el código anterior, se necesitaron más de 30 instrucciones SISD para realizar el producto punto, pero solo fueron requeridas 6 instrucciones SIMD para obtener el mismo resultado. ¡Impresionante! ¿No?, ¿Resulta importante aprender esta tecnología?, no olvide seleccionar mi curso de microprocesadores el próximo semestre.

La reducción en la cantidad de instrucciones es en sí una mejora importante, ya que libera al procesador del trabajo de decodificación de muchas instrucciones. Las instrucciones SIMD, activan la mayor cantidad de recursos de computación disponibles en el CPU con el menor número de instrucciones.

Puede aprender más acerca de estas tecnologías de computación en un curso de microprocesadores o bien, encontrar la información necesaria con los fabricantes de procesadores como Intel o AMD.

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Programación Paralela vs Programación Secuencial

El objetivo principal de la programación paralela, es la reducción del tiempo necesario para resolver determinado problema con un mayor volumen de datos. La ventaja de la programación secuencial radica en su facilidad de aprendizaje, análisis e implementación. Casi todos los cursos de programación, de hecho inician con el enfoque de programación secuencial para facilitar la comprensión de los conceptos fundamentales de la programación estructurada y la programación orientada a objetos.

En la sección anterior analizamos dos tipos de programación en la forma en la que se procesan los datos, por una parte escalar por escalar mediante la ejecución secuencial de instrucciones SISD; y por otra parte vector por vector mediante la ejecución secuencial de instrucciones SIMD.

Ahora bien, puede que surja la siguiente pregunta: ¿Hice paralelismo o no? La respuesta es: "Si", pero a qué nivel. Aclarando esto, SISD realiza paralelismo operando a nivel de bits, ya que los escalares se componen de bits. Por ejemplo un "long" está compuesto por 32 bits (una variable de tipo 'long' es un vector de 32 bits). SIMD, por otra parte realiza paralelismo operando a nivel de datos de tipo vector y un VECTOR4D es una estructura de cuatro flotantes, donde cada flotante es de 32 bits. Entonces siempre hemos realizado paralelismo de la forma SISD, SIMD con los OPERANDOS. ¿Y qué hay de las OPERACIONES (Instrucciones), se pueden paralelizar? La respuesta es también "Si", a veces.

Programación no es lo mismo que ejecución

A Programación nos referimos a la actividad que realiza un usuario programador (la persona que hace programas) y dicha actividad consiste en establecer en el computador, una "secuencia" de instrucciones de tal manera que corresponda a determinado algoritmo.

Algoritmo: Conjunto finito de operaciones ordenadas sobre operandos que resuelven un problema en tiempo finito.

Programa: Conjunto finito de operaciones ordenadas descritas en algún lenguaje formal de programación que corresponde a un algoritmo. Puede ser necesaria la traducción de un lenguaje formal para humanos hacia un lenguaje formal para CPU (Lenguaje Máquina).

La ejecución, es el trabajo asignado a un procesador de realizar las operaciones establecidas en un programa.

La ejecución corre por cuenta del CPU y si el procesador es superescalar entonces puede analizar el flujo de instrucciones de nuestros programas, analizar las dependencias entre los resultados de las instrucciones y en caso de que dos instrucciones sean independientes entre sí, estas se ejecutarán en paralelo.

Ejemplo: Suponga que se tiene un programa que calcula la siguiente asignación:

x=(a+b)*(c+d);

El procesador analiza el flujo de instrucciones y encuentra que la suma a+b se puede calcular en paralelo con c+d. Por lo que realiza paralelismo a nivel de instrucción en las sumas a+b y c+d. Pero es claro que no puede realizar producto en paralelo con las sumas ya que primero se deben calcular las sumas. El producto entonces espera a que esté disponible el resultado de las sumas.

La conclusión que hasta el momento podemos establecer es que los CPU's han evolucionado de tal manera, que gran parte de su electrónica está destinada al análisis del programa y flujo de datos para detectar independencias de ejecución y una vez encontradas estas independencias entre instrucciones, las ejecuta en paralelo.

El CPU puede ejecutar tantas instrucciones como recursos de computación existan en un núcleo de CPU. La cantidad de instrucciones que pueden ejecutarse simultáneamente en un CPU queda determinada en mayor medida a la disponibilidad de ALU's (Unidades aritmético-lógicas disponibles) y a la forma en la que un núcleo de CPU las administra y de la efectividad del analisis del flujo de instrucciones y detección de dependencias. Un núcleo de CPU moderno puede estar conformado por 2 o más ALU's por núcleo que pueden todas ellas operar en paralelo. Las ALU's pueden realizar sumas, restas, desplazamientos de bits y operaciones lógicas de bits, es decir operar con números enteros y sus bits. Si el CPU cuenta con la capacidad de ejecución de instrucciones SIMD, deberá contar con al menos dos ALU's SIMD para pode ejecutar dos o más instrucciones SIMD de manera concurrente.

¿Si el CPU en uno solo de sus núcleos hace tantas optimizaciones que promueven el paralelismo sin que yo modifique el programa, por qué tengo que aprender programación paralela? Tengo varias razones para ello:

1) Un solo núcleo optimizó tu programa en tiempos de ejecución: ¡Muy bien!, pero sólo lo hizo a nivel de instrucción tratando de ejecutar tantas instrucciones en paralelo como le sean posible de tu programa.
2) ¿Y los demás núcleos? ¡Bingo!, están de ociosos. ¡¿Por qué?! Porque solo hemos definido una tarea, como de costumbre en la programación secuencial convencional.

¿Se acuerdan de multitarea?, ¿Qué pasa si tengo más soluciones por ejecutar que procesadores disponibles? Efectivamente, recurrimos a la multitarea para distribuir el tiempo de ejecución. Es claro que la multitarea es controlada por el sistema operativo, por lo que será necesario aprender nuevos conceptos que facilitan el uso de los recursos de computación mediante el uso de éstos sistemas operativos multitarea.

Encontrando oportunidades de paralelismo a partir de la programación secuencial.

Definamos lo que entenderemos por programación paralela. Es la actividad del programador de definir secuencias de operaciones de tal manera que se maximice el uso de los recursos computacionales disponibles en un computador con el objetivo de reducir la cantidad de tiempo requerida para resolver un problema o procesar conjuntos de datos más grandes.

La programación paralela, por lo general se logra a través de la una estrategia de programación divide y vencerás. En donde un algoritmo es separado en algoritmos más simples e independientes entre sí, luego se definen sus correspondientes programas y a cada programa resultante se le asigna un procesador para que las ejecute. Estas soluciones más simples producen resultados parciales, que son luego mezclados para formar la solución completa. En resumen hay que seguir los siguientes pasos:

- Dividir la solución en soluciones más simples que no dependan entre sí. Este paso es uno de los más complejos ya que tenemos identificar las oportunidades de paralelismo al de verificar cualquiera de los siguientes criterios de independencia:
    - Existe Independencia entre dos porciones del algoritmo con respecto al orden de ejecución: Si dos diferentes porciones de un algoritmo pueden ejecutarse en cualquier orden sin afectar el resultado, entonces pueden ejecutarse en paralelo
    - Existe Independencia entre dos porciones del algoritmo con respecto a sus entradas y salidas de datos: Si las entradas de una porción del algoritmo no dependen de los resultados de una porción anterior, entonces ambas porciones pueden ejecutarse en paralelo.
- Mezclar los resultados parciales para obtener el resultado completo.

Terminaremos este fascículo con un problema con solución secuencial y lo paralelizaremos para comprender mejor el proceso de la programación paralela. El problema consiste en computar la suma de las cantidades almacenadas en un arreglo de 1024 elementos. Supongamos que se cuenta con un CPU con cuatro núcleos.

La solución secuencial se puede calcular así:

$$Suma = \sum_{i=0}^{1023} a_i$$

Como la suma es asociativa, podemos reescribir la anterior ecuación como:

$$Suma = \sum_{i=0}^{1023} a_i = \sum_{i=0}^{255} a_i + \sum_{j=256}^{511} a_j + \sum_{k=512}^{767} a_k + \sum_{l=768}^{1023} a_l$$

En donde cada sumatoria tiene sus propios índices. También se observa que los resultados parciales son independientes con respecto al orden en el que se calculen las sumatorias y no dependen las entradas de una suma parcial con el resultado de otra suma parcial.

Ahora bien, ya que pudimos separar la solución de la sumatoria en 5 soluciones más simples; en donde cuatro de éstas se pueden ejecutar en paralelo y una quinta que no puede hacerse hasta que la sumas por partes queden calculadas. La solución que no puede hacerse en paralelo solo realizaría tres sumas, lo cual hará muy rápido.

Modelo general de Computadores Paralelos SIMD

Ahora vamos a conocer los modelos computacionales de procesamiento paralelo que nos permitirán diseñar programas que efectivamente usen paralelismo a nivel de tarea. El modelo general de Computador Paralelo SIMD, es una relación entre los recursos de almacenamiento de datos, los recursos de procesamiento y las tareas.

Modelo general de un CPU

En la siguiente figura se muestran algunos de los elementos de procesamiento de un CPU moderno y los elementos de almacenamiento temporal de datos.

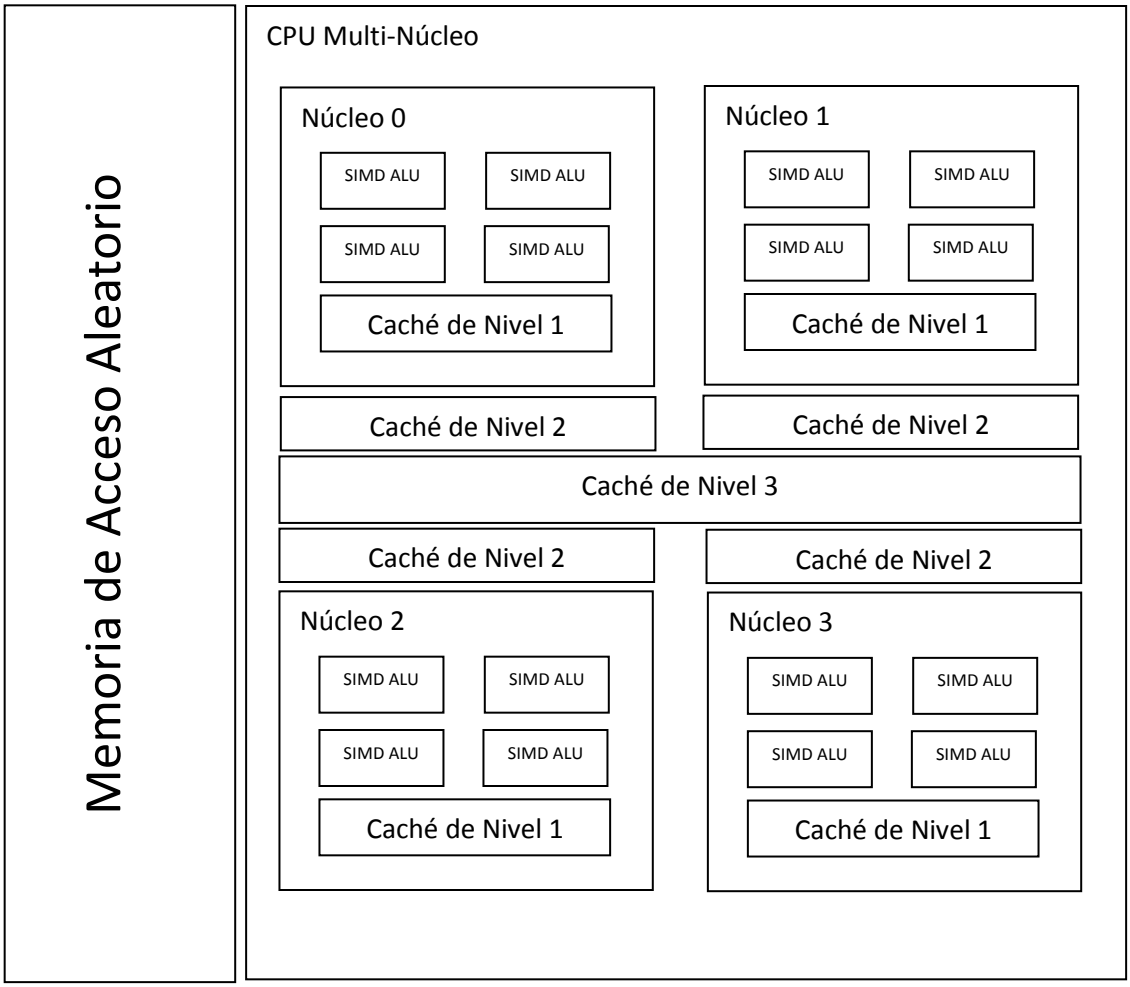


Ilustración 5 Componentes lógicos de un CPU en la que se ilustran las unidades de ejecución SIMD, cachés y sus núcleos.

Por lo general los CPU's están sincronizados a frecuencias de trabajo que varían entre 1GHz hasta 4GHz o más. Esto permite la ejecución de instrucciones de manera muy rápida. Además, una parte

de la electrónica se dedica al análisis del flujo de datos y detección de dependencia de instrucciones para promover el mayor paralelismo por núcleo.

La memoria RAM externa, o simplemente memoria de sistema, logra transferencias muy grandes de datos cuando los bloques de información son de gran tamaño. Solo es posible alcanzar altas tazas de transferencia de datos por unidad de tiempo cuando el procesador realiza transacciones grandes. Por qué menciono esto, bueno, la mayor parte de las operaciones con el CPU, se hacen con bloques relativamente pequeños, y cuando el procesador necesita actualizar la memoria de sistema debido a un fallo de caché, entonces el costo de iniciar una transacción a la memoria RAM por un volumen de datos pequeño, afecta al rendimiento ya que las transacciones están limitadas por la sincronía, la ejecución de la transacción, los ciclos de espera debido al refresco (DRAM) y la velocidad de operación de la memoria RAM. La frecuencia de operación de la RAM es mucho menor a la velocidad de operación del CPU.

Para aliviar los síntomas de bajo desempeño, el CPU se implementa con una variedad de cachés grandes y jerárquicos, los cuales, maximizan la reutilización mediante lectura y escritura temporal de datos, al mismo tiempo que minimizan la frecuencia de acceso a la memoria RAM externa que es masiva pero se vuelve lenta cuando se accede a datos con dimensión pequeña. Los algoritmos de caché que están implementados en cada uno de los elementos de caché, hacen que el CPU sea eficaz para manipular volúmenes de datos grandes y pequeños que pudieran estar dispersos en localidades de la memoria RAM.

La característica de alta eficiencia en acceso aleatorio de los CPU's hacia la RAM permite que se puedan definir estructuras complejas de acceso disperso o no contiguo de datos como lo son las listas, los árboles de búsqueda y clasificación, los grafos, tablas de dispersión, así como para estructuras contiguas de datos como el trivial arreglo o buffer. También beneficia al excelente desempeño de programas grandes y complejos, en donde los saltos y llamadas a funciones se encuentras dispuestas de manera dispersa en la memoria RAM, tal y como sucede con el código que conforma a las aplicaciones como las de diseño CAD, Juegos, Servicios, Gestores de Bases de Datos y el complejo Sistema Operativo mismo y el complejo proceso de comunicación entre éstas aplicaciones.

La diferentes formas en la que la memoria del sistema es utilizada por cada una de éstas aplicaciones que se ejecutan al mismo tiempo, ha orientado el diseño de los procesadores CPU principalmente para promover acceso a la RAM de datos cuya longitud es variable y generalmente dispersa.

En resumen:

- El CPU es eficaz al acceso aleatorio a datos en la memoria RAM, gracias a sus cachés.
- El CPU es eficiente en el manejo de datos pequeños (un byte por ejemplo) y dispersos en las localidades de la memoria RAM.
- EL CPU puede realizar tareas demasiado complejas y diferentes al mismo tiempo.

- El CPU tiene velocidades de acceso por transferencia de datos desde o hacia la memoria RAM relativamente lentas, ya que promueve el acceso disperso en vez del acceso masivo en transacciones de gran volumen.
- Los CPU tienen pocos núcleos debido a la complejidad de integración (cantidad de transistores muy, muy rápidos) que implementan muchas facilidades (cantidad de instrucciones), formas de operar y acceder a la información y por supuesto la optimización en tiempo real del código.
- El CPU tiene conectada una RAM de dimensiones masivas. ¡Mucho espacio para programas y datos en RAM!
- El CPU es fácil de programar y de depurar, ya que tenemos a un sistema operativo complejo con las operaciones que facilitan éste proceso.
- El CPU es muy costoso. Debido a que utiliza alta frecuencia y los transistores más modernos en gran cantidad para su funcionamiento e ingeniería de optimización y ejecución en tiempo real de código.
- El número de tareas que se pueden ejecutar simultáneamente en un CPU, está limitada por el reducido número de núcleos.
- Los CPU's presentan valores de consumo de corriente relativamente altos y bajas tasas de rendimiento por unidad de potencia TFLOP/Watt, si se les compara con los GPU.

Modelo abstracto de planificación y ejecución multitarea mediante procesos e hilos.

Los sistemas operativos multitarea concurrentes como Linux y Windows nos dan las herramientas para que podamos ejecutar y administrar con relativa facilidad programas que se ejecuten en paralelo y nos permiten compartir el CPU con otras aplicaciones de forma transparente. Los sistemas operativos de ésta categoría se encargan de la administración de los recursos de memoria, dispositivos, procesador y de la asignación de recursos de ejecución a las tareas. La asignación de recursos de ejecución a tareas debe ser progresiva, es decir, eventualmente todas las tareas deben ejecutarse. La asignación de ejecución también debe ser justa entre las tareas con la misma prioridad, la propiedad de justicia indica al planificador de tareas que deberá compartir el procesador en tiempos iguales entre las tareas.

Tanto Windows como Linux, son sistemas operativos multitarea con prioridad, en donde a cada tarea se le asigna un nivel de prioridad. El nivel de prioridad se define mediante un número entero cuya enumeración especifica el sistema operativo particular.

El sistema operativo es el responsable de iniciar, ejecutar y finalizar las tareas, planificar su ejecución considerando la multitarea, control de acceso concurrente a recursos y la asignación de tareas a procesadores (núcleos) para la ejecución concurrente y finalmente administrar la asignación de recursos de memoria y dispositivos a las tareas que así lo requieran. Linux y Windows ofrecen ésta funcionalidad de hecho es la más importante.

Modelo de administración de procesos, planificación y ejecución de hilos en Windows

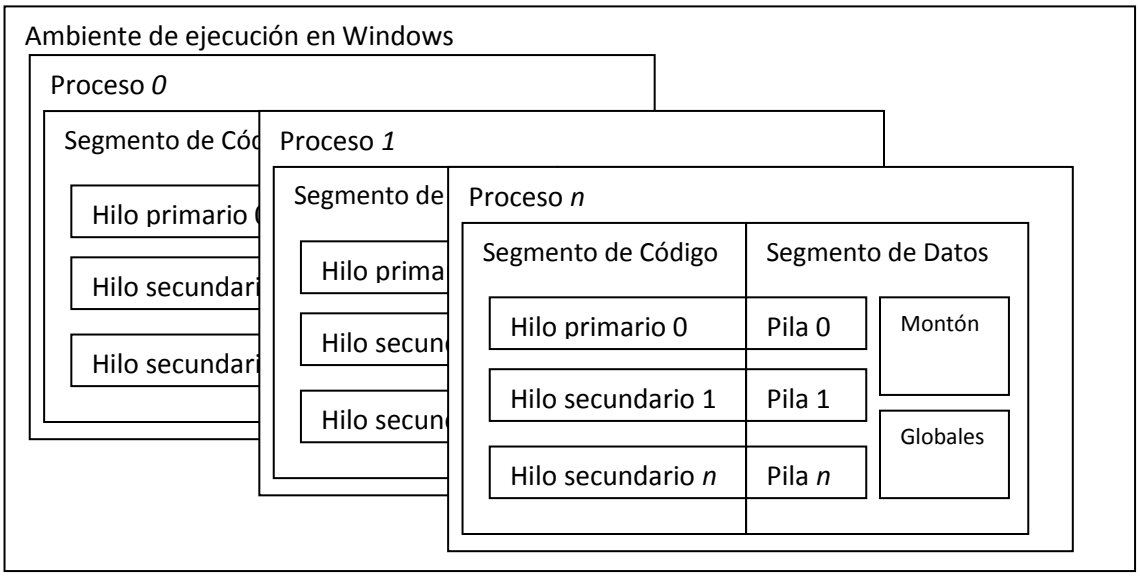
En este curso utilizaremos la API de Windows para desarrollar aplicaciones con potencial de computación paralela. Sin embargo, para casi cualquier función de la API de procesos de Windows, existe una función equivalente o compuesta en Linux, y viceversa.

Definición de proceso e hilo en Windows

Ya estamos a punto de iniciar el desarrollo de aplicaciones multitarea, para ello, hay que conocer el entorno de ejecución de hilos de Windows.

Un proceso es un conjunto de hilos que comparten el mismo segmento de datos y de código. Cada hilo tiene su segmento de pila para la administración de variables locales. Cada proceso posee un nivel de prioridad base de ejecución de sus hilos que se le conoce como *Prioridad de Clase o Prioridad Base* de Hilo. Un proceso no comparte sus espacios virtuales de direcciones (pilas, datos y código) con otros procesos.

Un hilo, es la unidad de tarea más simple que puede ser planificada y ejecutada. Cada hilo tiene su propio segmento de pila, pero comparte el segmento de código y de datos con otros hilos pertenecientes al mismo proceso que éste hilo. Un hilo siempre pertenece a un proceso y solo a uno. Cada hilo posee un nivel de prioridad de ejecución llamada *Prioridad de Hilo*.



La prioridad efectiva de ejecución o simplemente prioridad de ejecución, se compone de la Prioridad Base de Hilo más la Prioridad de Hilo. Entre más grande sea, más prioridad de ejecución tiene sobre otros hilos. Eventualmente todos los hilos serán ejecutados, por lo que Windows es progresivo. Se prevé justicia de ejecución entre hilos de la misma prioridad efectiva incluso entre

hilos de otros procesos que tengan el mismo nivel. La siguiente figura describe esta relación y agrupación entre procesos, hilos y memoria.

Windows sólo ejecuta y planifica hilos, más no procesos. Recordar que un proceso tiene al menos un hilo.

En particular cada proceso tiene un hilo especial identificado como *Hilo Primario*. El hilo primario puede crear hilos secundarios y estos a su vez de manera recurrente. Si el hilo primario termina su ejecución, entonces el proceso es destruido y se le desalojan los recursos de memoria y dispositivos. El proceso también será destruido incluso si hay hilos secundarios en ejecución, los cuales serán terminados de forma inesperada. ¡Hay que cuidar el hilo primario! Solo hay dos tipos de hilos, un solo hilo primario y cualquier número de hilos secundarios.

En el siguiente fascículo, descubriremos como se implementa una aplicación multitarea concurrente en términos de Windows. Estudien e investiguen.

Programación Multitarea en CPU

En esta parte, aprenderemos cómo organizar nuestra solución para aprovechar las facilidades de multitarea y activación del paralelismo que ofrece Microsoft Windows. Como toda nueva técnica de computación, la programación en paralelo puede ofrecernos nuevas oportunidades para resolver problemas más rápido, pero también surgen otros interesantes problemas debido a la concurrencia. Estos problemas de concurrencia están relacionados con la administración de hilos y el acceso a datos que debemos enfrentar para diseñar la mejor estrategia de implementación. El sistema operativo nos entrega un conjunto de objetos y funcionalidad que nos ayudará a controlar la multitarea y el paralelismo para lograr nuestro objetivo de aceleración del procesamiento de manera consistente, estable, progresiva y justa.

Al terminar esta parte del curso, habrá desarrollado criterios básicos para justificar el empleo de técnicas de procesamiento paralelo en CPU. También tendrá las habilidades básicas para diseñar arquitecturas de aplicaciones multitarea y de procesamiento paralelo utilizando las facilidades que el sistema operativo tiene que ofrecer para éste fin. Conocerá cómo administrar y utilizar los recursos de procesamiento disponible a través de la creación y ejecución controlada de múltiples hilos, estableciendo estrategias eficaces y eficientes de comunicación entre éstos, así como la implementación de acceso a datos de forma concurrente, correcta, estable y consistente.

Multitarea, paralelismo y acceso a recursos compartidos entre hilos

La multitarea en el sistema operativo Windows, se hace a través de dos abstracciones muy importantes: el proceso y el hilo. El kernel de Windows, se encarga de planificar y asignar hilos activos de cualquier proceso para su ejecución en cualquiera de los núcleos disponibles en el CPU. Por lo que el diseño de aplicaciones que necesiten paralelismo a nivel de tarea se simplifica en gran medida.

Justamente a través del concepto de hilo, nuestro esfuerzo de programación se concentra más en la implementación de una actividad particular, en vez de pensar en cómo se lleva a cabo el paralelismo o la multitarea en nuestro computador. Sin embargo, no todo es tan simple ya que el sistema operativo no conoce los detalles de implementación de nuestro programa y tampoco las estructuras de datos que se utilizan en ese mismo programa. Debido a esto, Windows no puede decidir de manera automática cuando iniciar y detener los hilos de tal manera que no existan conflictos de acceso a las estructuras de datos en caso de que más de una tarea accede a dicha estructura. A esta situación, en la que una o más estructuras de datos son compartidas entre dos o más tareas, se le conoce como concurrencia de acceso a datos. La concurrencia de acceso a datos y su control es un tema importante, ya que de no implementar control, las tareas pueden dañar la integridad estructural de la información por falta de coordinación.

Por ejemplo, supongamos que se tiene una lista enlazada y queremos que dos tareas iteren su contenido para leer y realizar distintos cómputos. Es claro, que ambas tareas, no modifican el

contenido de la lista, por lo que no existe el potencial de dañar la información en dicha lista si ambas tareas leen la lista al mismo tiempo, incluso en partes diferentes de la misma. Ahora supongamos que existe un tercer hilo, cuya tarea consiste en modificar el contenido de esa misma lista, ya sea insertando o eliminando elementos. ¿Cómo saben los hilos lectores que están accediendo a la información actualizada y válida desde el hilo que la modifica?, ¿Cómo asegurarnos que los iteradores son consistentes y vigentes ante tal acceso concurrente? Si no se implementa una estrategia de acceso a datos, esto puede conducir a un funcionamiento errático, inconsistente e inestable (¡PUM!).

Para explicarlo mejor, suponga que tenemos un tren en movimiento a través de las vías férreas. En este caso el tren es la tarea y las vías férreas son las estructuras de datos. ¿Qué pasaría si una actividad de mantenimiento sobre las vías del tren, en este caso, una segunda tarea, se inicia en cualquier momento sin previo aviso y sin control? Pueden suceder varias cosas, unas buenas y otras malas. Suponga pues, una situación en la que el tren ya ha pasado por el segmento de vías férreas a las cuales se les dará mantenimiento. Bajo esa condición, el mantenimiento no afecta el movimiento del tren y no ocurre ningún inconveniente. Una segunda situación puede ser que el tren está pasando por el segmento de vías que se les está dando mantenimiento sin previo aviso al tren. Lo único que se puede inferir de ésta última situación es un desastre muy lamentable. Como puede darse cuenta, a un segmento de vías férreas o se les da mantenimiento o son usadas por el tren para desplazarse, pero nunca al mismo tiempo si es que no queremos que pase nada malo. En este caso, el uso del segmento de vías férreas que puede ser por el tren o por el equipo de mantenimiento, simplemente no debe ocurrir al mismo tiempo. Un acceso de éste tipo, en donde dos o más tareas no deben acceder el mismo recurso simultáneamente se le conoce como exclusión mutua, tema que analizaremos más adelante.

Las situaciones de conflicto de acceso a recursos, como la que se trató en el párrafo anterior, son evidentes y fáciles de identificar cuando hablamos de paralelismo. Pero ¿Qué hay de la multitarea?

Ya sabemos que multitarea no implica necesariamente paralelismo total. Es claro en el caso de un CPU mono-núcleo. El sistema operativo recurre a la multitarea, para compartir el tiempo de ejecución del procesador en caso de que el número de tareas activas superen el número de núcleos del CPU. Bajo situaciones de conmutación de tarea en uno o más núcleos, el problema de los trenes también puede suceder. Supongamos que el tren está pasando por un segmento de vías y le tomamos una instantánea. Luego cambiamos a la tarea de mantenimiento con la casualidad de realizar cambios sobre ese mismo segmento de vías donde está el tren. Luego tomamos otra instantánea en la que el mantenimiento de las vías no haya concluido, para luego cambiar a la tarea del tren en movimiento. ¿Será seguro que el tren continúe su desplazamiento a través de un segmento de vías cuyo mantenimiento no ha concluido? La respuesta es: no y no me gustaría viajar en ese tren.

Windows se encarga de la multitarea y del multiprocesamiento concurrente, lo cual hace muy bien. Windows también nos puede servir para controlar la ejecución de tareas y el control de

acceso a recursos, impidiendo o permitiendo la ejecución de hilos en determinados puntos de control que determinemos como seguros y así garantizar la estabilidad y consistencia en la forma en la que procesamos datos compartidos entre tareas.

Prestaciones de un sistema operativo para el desarrollo de aplicaciones multitarea y multiprocesamiento

En el caso particular de Windows, podemos establecer los siguientes conjuntos de funcionalidad para el desarrollo de aplicaciones multitarea y control del multiprocesamiento o paralelismo.

- Funciones de consulta de la infraestructura de procesamiento disponible en el CPU: Las cuales nos permitirán tomar decisiones durante la inicialización de nuestro programa para definir el número de hilos adecuado para maximizar el paralelismo de tarea.
- Funciones de creación de procesos e hilos: Estas funciones nos permiten iniciar procesos a partir de archivos ejecutables o bien crear hilos dentro del proceso.
- Funciones de creación y liberación de objetos de sincronización, los cuáles nos permitirán implementar control de estado de ejecución de los hilos como puede ser la Mutex, Sección Crítica, Eventos, Semáforos.
- Funciones de Espera. Las funciones de espera, le indican al planificador de Windows, que mientras los objetos de sincronización o de señalización no estén liberados, la ejecución del hilo se debe suspender hasta que los objetos queden liberados o señalizados. Durante la suspensión no se le asigna ejecución a ese hilo en espera.
- Funciones atómicas. Este tipo de funciones permiten realizar una operación aritmética simple, como la comparación, intercambio, incremento o decremento sobre una o más variables con acceso de exclusión mutua entre hilos.
- Funciones para establecer canales de comunicación entre procesos. La comunicación entre hilos de un mismo proceso resulta simple ya que comparten el mismo espacio para datos. Sin embargo, para compartir datos entre procesos será necesario utilizar recursos administrados por el sistema operativo para realizar dicha funcionalidad. Entre los objetos que se pueden utilizar son, los tubos (pipes), sockets de red y mapeos por archivos.

Creación de hilos en Windows

Para crear hilos secundarios en Windows, se utiliza la función *CrateThread*, la cual pide como parámetros un puntero a una función que se encargará de realizar la tarea y un puntero void* con el parámetro de inicio. La función *CreateThread*, puede crear un hilo inicialmente suspendido o en ejecución. Si el hilo es creado exitosamente, la función *CreateThread* retorna un manipulador de hilo, la cual nos permitirá administrar dicho hilo. Cuando un hilo se inicia en suspensión, debe reasumirse la ejecución invocando a la función *ReasumeThread* indicando su manipulador.

La implementación de un hilo en Windows, escrito en C, debe respetar el siguiente prototipo:

```
DWORD WINAPI ThreadProc(void* pInitialParams);
```

En donde "ThreadProc" es el identificador de hilo definido por el usuario. El parámetro pInitParams se especifica al momento de crear el hilo. El convenio de llamada WINAPI, le indica al compilador que la función será invocada por el sistema operativo cuando se le asigne un hilo. Los procedimiento de hilo, no deben invocarse directamente por el programa, ya que de hacerlo, se comportarían como funciones bajo la ejecución del hilo que lo invoca. Un hilo retorna un valor de tipo DWORD (unsigned long) para indicar el estado de terminación del hilo. Un hilo debe retornar 0, si la terminación del hilo fue normal. Un valor distinto de cero para indicar terminación de la tarea con excepciones o errores durante la realización de la misma.

El siguiente ejemplo de programa nos muestra cómo crear un hilo secundario al cual le pasamos parámetros de inicio para que sean procesados por los hilos recién creados.

```
//Estructura personalizada de parámetros del hilo
struct THREADPARAMS
{
	int x;
	float y;
	char* psz;
};

//Implementación del hilo secundario
DWORD WINAPI ThreadProc(void* pInitParams)
{
	THREADPARAMS* pParams=(THREADPARAMS*)pInitParams;
	//Hacer alguna tarea....
	.
	.
	.
	//Al terminar, liberar memoria de los parámetros
	delete pParams;
	return 0;
}

//Hilo primario
int _tmain(int argc, _TCHAR* argv[])
{
	//El hilo primario inicia un hilo secundario y le pasa parámetros
	THREADPARAMS* pParams=new THREADPARAMS;
	pParams->x=10;
	pParams->y=3.141592f;
	pParams->psz="Hola Mundo";
	CreateThread(NULL,4096,ThreadProc,pParams,0,NULL);
	.
	.
	.
	return 0;
}
```

En el programa anterior, el hilo primario crea un hilo indicando una pila inicial de 4096 bytes, cuya tarea está especificada por la función "ThreadProc", con parámetros THREADPARAMS, que planifica para su inmediata ejecución.

Los parámetros se pasan al hilo secundario a través de memoria de montículo o montón (heap). Para ello asignamos los recursos de memoria haciendo uso del operador “new”. Cuando Windows ejecuta el hilo, le pasa una copia del puntero para que tenga acceso los parámetros que el hilo primario colocó y pueda comenzar la tarea parametrizada. Puede observarse que el hilo secundario, libera los recursos asignados a los parámetros antes de que el hilo termine utilizando el operador “delete”. El montículo (heap) implementa acceso de exclusión mutua, por lo que las asignaciones y las liberaciones son estables entre hilos. Esta simple técnica de paso de parámetros a hilos secundarios es muy simple de hacer y de mantener.

Nota: Solo las asignaciones y las liberaciones de memoria gozan de exclusión mutua en el árbol del montículo de asignación dinámica pero el acceso a los bloques de memoria debe sincronizarse en caso de que el bloque sea utilizado por más de un hilo.

Señalización entre hilos mediante Eventos

Hasta el momento hemos iniciado hilos sin considerar el momento en que éstos se ejecutan o sin la precaución de cuando éstos terminan. Esto puede conducir a serios problemas de sincronización y estabilidad. Windows, como ya se ha mencionado antes, solo planifica y ejecuta hilos, más no controla de manera implícita el acceso a datos. En la mayoría de los casos, el desarrollador de la aplicación debe instruir a Windows, cuándo es el momento adecuado para que los hilos secundarios comiencen a realizar determinadas tareas, en lo que el hilo primario prepara sus parámetros o volúmenes de datos que van a procesar. También será necesario conocer el momento en que cada una de las tareas terminen para poder recoger los resultados que éstos generen y producir el resultado total.

Los eventos son objetos de señalización que permiten identificar sucesos en determinados puntos de ejecución de cualquier hilo. Los hilos “señalizan” el evento para notificar a otros hilos que ha llegado su ejecución a cierto punto, que consideran seguro para que otros hilos realicen actividades de manera consistente y estable.

Los eventos pueden imaginarse como banderas que son administradas por el sistema operativo, que le permiten al planificador de tareas, suspender o iniciar la ejecución de otros hilos que esperan un evento. Los hilos que “esperan” un evento, se encuentran suspendidos hasta que la señal se active. Esto permite implementar técnicas de control de ejecución y así evitar situaciones de carrera o concurrencia inestable.

Los eventos pueden estar en uno de dos estados: Señalizados (Establecido, Activado, ON, 1, TRUE) o No Señalizado (Restablecido, Desactivado, OFF, 0, FALSE).

Existen dos modalidades de evento: Los de reinicio manual (biestable) y los de reinicio automático (monoestable).

Los eventos de reinicio automático, vuelven al estado desactivado cuando un hilo que se encontraba esperando dicho evento se activa. Esto permite la puesta en ejecución de hilo a la vez. Por ejemplo, suponga, que tres hilos se encuentran esperando un evento de reinicio automático. Cuando otro cuarto hilo señaliza el evento, uno de los tres hilos que se encontraban esperando comienza su ejecución. Para que todos los hilos comiencen su ejecución será necesario señalizar tres veces el mismo evento.

Los eventos de reinicio manual, conservan el estado de señalización que se les indique. Por lo que si varios hilos se encontraban esperando a un evento de reinicio manual, entonces todos los hilos inician la ejecución inmediatamente con tan solo establecer el evento una vez. Si el evento sigue señalizado y otro hilo intenta esperar dicho evento, entonces la ejecución continúa sin esperar. El evento debe reiniciarse manualmente para que los hilos esperen. Esto se utiliza para que se inicie la ejecución de varios hilos con tan solo una activación.

La activación o puesta en 1 de un evento se hace a través de la función *SetEvent* que recibe el manipulador de un evento a establecer.

La desactivación o simplemente reinicio se hace a través de la función *ResetEvent* que recibe el manipulador de evento a reiniciar.

Para que un hilo espere a que se señalice o se active un evento, deberá invocar a la función *WaitForSingleObject* indicando el manipulador de evento a esperar y especificando cuánto tiempo máximo debe esperar.

Para que un hilo espere a que se activen uno o más eventos, deberá invocar a la función *WaitForMultipleObjects* indicando el arreglo de manipuladores de eventos a esperar, el número de eventos en el arreglo de eventos a esperar, si debe o no esperar a que todos los eventos se señalicen y cuánto tiempo máximo debe esperar.

Para resolver el problema de iniciar tareas y conocer cuando las éstas terminan, podemos recurrir a un objeto de sincronización de tipo evento.

El siguiente ejemplo muestra como iniciar dos hilos secundarios al mismo tiempo y nos permite saber cuándo éstos terminan para finalizar de manera segura el hilo primario:

Programa 1:

```
// Multithreading01.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <Windows.h>
#include <iostream>
using namespace std;

//Estructura de paso de parámetros entre hilos
struct THREADPARAMS
{
          HANDLE hEventStart;   //Hilo espera orden de inicio
          HANDLE hEventFinished;          //Hilo notifica su finalización
};
//Tipos de Hilos secundarios Clase A y los Clase B (protitipos)
DWORD WINAPI ThreadClassA(void*);
DWORD WINAPI ThreadClassB(void*);
```

```
//Hilo primario (Siempre lo ha sido... main...)
int _tmain(int argc, _TCHAR* argv[])
{
	//Parámetros de las tareas
	THREADPARAMS* pParams;
	//Crear los objetos de sincronización
	//Crear eventos, de tipo de reinicio manual, con estado inicial no señalizado (false)
	HANDLE ahEventFinished[2];
	ahEventFinished[0]=CreateEvent(
		NULL,	//Seguridad heredada del hilo primario
		TRUE,	//Reinicio manual, las transiciones de 1 a 0 se hace manualmente
		FALSE,	//Estado inicial NO SEÑALIZADO = FALSE = 0 = OFF
		NULL	//Evento anónimo
		);
	ahEventFinished[1]=CreateEvent(NULL,TRUE,FALSE,NULL);
	HANDLE hEventStart=CreateEvent(NULL,TRUE,FALSE,NULL);
	//El hilo primario inicia dos tareas distintas en un hilo cada una con los eventos que deben usar
	pParams=new THREADPARAMS;
	pParams->hEventFinished=ahEventFinished[0];		//Hilo notifica su finalización con este evento
	pParams->hEventStart=hEventStart;				//Hilo espera ordenes de inicio con este evento
	CreateThread(NULL,4096,ThreadClassA,pParams,NULL,NULL);

	pParams=new THREADPARAMS;
	pParams->hEventFinished=ahEventFinished[1]; //Hilo notifica su finalización con este evento
	pParams->hEventStart=hEventStart;				//Hilo espera ordenes de inicio con este evento
	CreateThread(NULL,4096,ThreadClassB,pParams,NULL,NULL);

	cout<<"Presione <enter> para permitir la ejecución de los hilos secundarios que ya se encuentran listos...\n";
	cin.ignore();

	//Comienza la ejecución de todos los hilos esperando a que éste evento se señalice.
	SetEvent(hEventStart);

	//Esperar a que los hilos secundarios terminen su trabajo
	cout<<"Hilo primario se encuentra esperando a que los hilos secundarios terminen\n";
	WaitForMultipleObjects(
		2,		//Esperar a dos eventos
		ahEventFinished,	//Arreglo con los eventos a esperar
		TRUE,	//Esperar a que todos queden señalizados
		INFINITE	//Hasta que terminen
		);
	cout<<"Las tareas han finalizado exitosamente. El hilo primario puede salir de manera segura.\n";
	//Desalojar recursos de sincronización
	CloseHandle(ahEventFinished[0]);
	CloseHandle(ahEventFinished[1]);
	CloseHandle(hEventStart);
	return 0;
}

//Implementación del hilo A
DWORD WINAPI ThreadClassA(void* pInitParams)
{
	THREADPARAMS* pThreadParams=(THREADPARAMS*)pInitParams;

	//Esperar la orden para comenzar
	cout<<"Hilo Clase A Listo para comenzar la ejecución\n";
	WaitForSingleObject(pThreadParams->hEventStart,INFINITE);

	cout<<"Hola mundo soy el hilo de Clase A\n";
	//Hacer algo de trabajo para mantener ocupado a un Núcleo
	for(volatile long long i=0;i<1024*1024*512;i++);

	cout<<"La tarea A ha terminado\n";
	//Extraer una copia del evento hEventFinished antes de liberar la estructura de parámetros
	HANDLE hEventFinished=pThreadParams->hEventFinished;
	//Liberar parámetros
	delete pThreadParams;
	//Notificar que éste hilo ha terminado
	SetEvent(hEventFinished);
	return 0;
}

//Implementación del hilo B
DWORD WINAPI ThreadClassB(void* pInitParams)
{
	THREADPARAMS* pThreadParams=(THREADPARAMS*)pInitParams;
	//Esperar la orden para comenzar
	cout<<"Hilo Clase B Listo para comenzar la ejecución\n";
	WaitForSingleObject(pThreadParams->hEventStart,INFINITE);

	cout<<"Hola mundo soy el hilo de Clase B\n";
	//Hacer algo de trabajo para mantener ocupado a un Núcleo
	for(volatile long long i=0;i<1024*1024*256;i++);

	cout<<"La tarea B ha terminado\n";
	//Extraer una copia del evento hEventFinished antes de liberar la estructura de parámetros
	HANDLE hEventFinished=pThreadParams->hEventFinished;
	//Liberar parámetros
	delete pThreadParams;
	//Notificar que éste hilo ha terminado
	SetEvent(hEventFinished);
	return 0;
```

```
}
```

En el ejemplo anterior se utilizan tres objetos de sincronización de tipo Evento. Uno de ellos indica la orden de inicio para que los hilos secundarios comiencen su tarea. Al principio cada hilo secundario se encuentra esperando un simple objeto de sincronización, pero como el objeto de sincronización no ha sido señalizado (activado), entonces el planificador suspende la ejecución de los hilos secundarios. Cuando el usuario presiona la tecla <enter> entonces, el objeto de sincronización es señalizado e inmediatamente el planificador reasume la ejecución de los hilos que se encontraban esperando. El hilo primario entra entonces en espera a que todos los eventos de finalización se activen, pero como los hilos no han terminado, se suspende su actividad. Cuando los hilos secundarios terminan, señalizan los eventos de finalización. Como el hilo primario estaba esperando a que todos los eventos de finalización se señalicen, el hilo primario reasume su actividad, finalizando de manera segura.

Actividades:

- Implemente el programa anterior, analícelo, compréndalo y experimente.
- Modifique el programa para permitir que solo se active un hilo a la vez cada vez que señalice hEventStart. Esto se puede hacer, pidiendo al usuario que presione dos veces la tecla <enter> y señalizando el evento de inicio. Deberá cambiar el tipo del evento a reinicio automático.
- Puede encontrar más información y forma de operación de todas las funciones de sincronización utilizadas en MSDN.
- Realice preguntas en caso de dudas en el grupo de Facebook.

Ejemplo de multiprocesamiento básico con información de acceso no ordenado

En los siguientes ejemplos, nos familiarizaremos con las técnicas básicas de procesamiento paralelo. Para ello, utilizaremos el siguiente el conjunto pretexto de datos de una malla geométrica "Monkey" compuesta de dos conjuntos de datos: Una colección de vértices y una colección de índices.

Cuando construya sus propias soluciones de procesamiento paralelo, es importante que conozca la estructura de la información que va a procesar, así como de las relaciones entre los diversos conjuntos de esa información, tal y como lo hace en la programación secuencial convencional. Si usted dispone de un buen conocimiento o documentación en las estructuras de datos que manipula, deberá encontrar propiedades que promuevan el procesamiento paralelo, de tal manera que se conserve la integridad referencial, la consistencia y la validez de los resultados.

Para comprender el tipo de información pretexto que utilizaremos, las colecciones de vértices e índices pueden visualizarse en una proyección de R^3 a R^2 así:

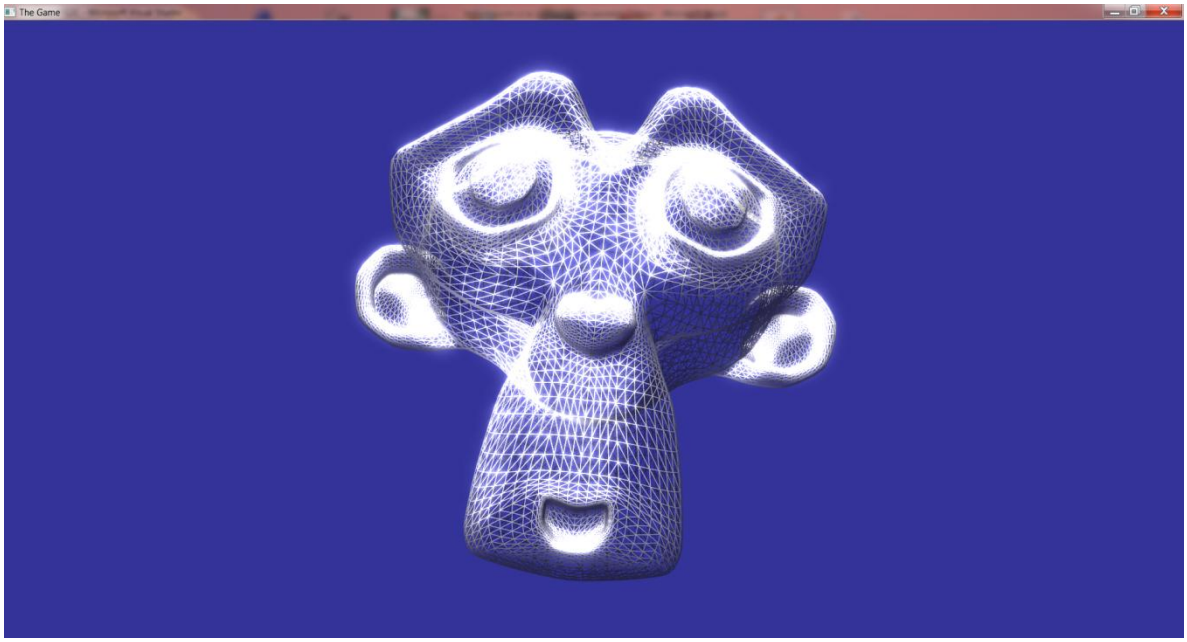


Ilustración 6 Proyección de triángulos a partir de la relación de vértices e índices de la malla Monkey.

El conjunto de vértices tiene elementos del conjunto R^4, es decir, elementos vectoriales [x, y, z, w], en donde x, y, z, w son elementos del tipo flotante que aproximan a los elementos de R (números reales). Las coordenadas están en el espacio homogéneo, de tal manera que el cambio de espacio de R^4 a R^3 queda definido como: [x,y,z,w] -> [x/w, y/w, z/w]. La componente 'w' se utiliza como componente de perspectiva definida por una función de transferencia w=f(z), siendo z la produndidad. Para la colección de vértices de Monkey que trataremos, la componente w=1, por lo que [x, y, z, 1] en R^4 es simplemente [x, y, z] en R^3. La imagen anterior, se logró a través de una transformación que modifica las componentes {x, y, z, w} tal que w=f(z) y w!=0. Otra ventaja

importante de la representación homogénea es que permite realizar transformaciones que unifican en una sola matriz de transformación a rotaciones, escalaciones y traslaciones. A éstas transformaciones que combinan a una transformación lineal con una función de traslación se le conocen como transformaciones afines. Pongo a su alcance esta importante información si desea explorar este fascinante tema. En estos ejercicios de cómputo paralelo, no realizaremos tal transformación y reducción de R^4 a R^2, solo queda para fines informativos y despertar curiosidad e interés.

El volumen de datos del conjunto de vértices está almacenado en el archivo "Monkey.vertex" que puede descargar del grupo en Facebook. Los datos se encuentran en formato binario para conservar la mayor fidelidad de los vértices y evitar traducciones con pérdida de información. Los primeros cuatro bytes que aparecen al comienzo del archivo indican el número de vértices en el archivo (unsigned long). Luego, aparece el vector con los vértices. Cada vértice es un vector tetradimensional de la forma [x, y, z, w], en donde cada elemento es un flotante simple (float) de cuatro bytes. Por lo que cada vértice es de 16 bytes. Los vértices se encuentran en coordenadas homogéneas.

La colección de índices está almacenada en el archivo "Monkey.index" que también puede descargar el grupo en Facebook. El archivo contiene los índices sobre el arreglo de vértices, que definen las aristas de los triángulos de la malla. Al inicio del archivo se especifica un entero sin signo (unsigned long) de 4 bytes con el número de índices que indexan el vector de vértices. Inmediatamente después le sigue el vector de índices. Cada índice es un entero sin signo (unsigned long) de 4 bytes. El índice, indexa al vector de vértices. Cada triada ordenada de índices, indexa a tres vértices, conformándose así un triángulo. La cantidad de triángulos totales es el número de índices dividido entre tres.

El siguiente fragmento de programa muestra como cargar y estructurar la información de vértices e índices en la memoria RAM proveniente de los archivos correspondientes. Es claro que usted deberá enfrentarse a diversas fuentes de información como bases de datos, flujos de audio o vídeo con diferentes necesidades de almacenamiento, formato y algoritmos de procesamiento, etc., por lo que nos limitaremos a preparar la información de nuestro ejemplo pretexto.

```
#include <vector>
#include <fstream>
using namespace std;

struct VECTOR4D
{
	float x,y,z,w;
};

bool LoadVertexData(char* pszFileName,vector<VECTOR4D>& vOutVertexData)
{
	unsigned long ulVertexCount;	//Número de vértices a cargar
	fstream VertexFile;		//Archivo con el volumen de datos.
	//Abrir archivo en modo lectura y en modo binario
	VertexFile.open(pszFileName,ios::in|ios::binary);
	if(!VertexFile.is_open())
```

```
		return false;	//El archivo especificado no pudo ser abierto
	//Leer el número de vértices
	VertexFile.read((char*)&ulVertexCount,sizeof(unsigned long));
	//Asignar los recursos de almacenamiento en RAM.
	vOutVertexData.resize(ulVertexCount);
	//Leer los vértices en una sola ráfaga de lectura.
	VertexFile.read((char*)&vOutVertexData[0],ulVertexCount*sizeof(VECTOR4D));
	return true; //La información ya está en vOutVertexData
}
bool LoadIndexData(char* pszFileName,vector<unsigned long>& vOutIndexData)
{
	unsigned long ulIndexCount;//Numero de índices a cargar
	fstream IndexFile;		//Archivo con el volumen de datos.
	//Abrir archivo en modo lectura y en modo binario
	IndexFile.open(pszFileName,ios::in|ios::binary);
	if(!IndexFile.is_open())
		return false; //El archivo especificado no pudo ser abierto
	//Leer el número de indices
	IndexFile.read((char*)&ulIndexCount,sizeof(unsigned long));
	//Asignar los recursos de almacenamiento en RAM.
	vOutIndexData.resize(ulIndexCount);
	//Leer los índices en una sola ráfaga de lectura.
	IndexFile.read((char*)&vOutIndexData[0],ulIndexCount*sizeof(unsigned long));
	return true; //La información ya está en vOutIndexData
}
```

Ahora que disponemos de la información de la malla de Monkey en la memoria RAM, podemos procesarla de muchas maneras para obtener diversos resultados interesantes y útiles en el campo de la geometría.

Los siguientes casos de procesamiento paralelo, consideran situaciones en el que el orden de acceso (ya sea lectura o escritura) a los datos no importa, ya que no afectaremos a la integridad referencial y a la validez de los resultados bajo esta forma de acceso. A este tipo de acceso se le conoce como “Acceso no Ordenado (Unordered Access)”. Bajo la situación de acceso no ordenado, el concepto de exclusión mutua no se presenta. Eventualmente analizaremos situaciones de exclusión mutua.

Cómputo del centroide geométrico de una malla en base a posición de sus vértices.

El cetroide de una malla en términos de la posición de sus vértices se calcula así:

Si V es un vector de tamaño N, con vértices en R^4 con componente w=1, entonces el centroide puede computarse como:

$$Centroide = \frac{1}{N}\sum_{i=0}^{N-1} V_i$$

Analizando la ecuación anterior, podemos darnos cuenta que se realizan n sumas vectoriales y una sola división. La cantidad de sumas, en el caso de una malla, pueden ser miles o millones, dependiendo de la complejidad.

Ahora, vamos a suponer que disponemos de un computador con P-núcleos. El problema puede dividirse en tareas más simples aprovechándonos de la propiedad asociativa de la suma y asignando cada sumatoria a un procesador así:

$$Centroide = \frac{1}{N}\left(\sum_{i_0=0}^{a_0-1} V_{i_0} + \sum_{i_1=a_0}^{a_1-1} V_{i_1} + \cdots + \sum_{i_{P-1}=a_{P-2}}^{a_{P-1}-1=N-1} V_{i_{P-1}}\right)$$

Dónde:

$$0 < a_o < a_1 < \cdots < a_{P-1} = N$$

De tal manera que tenemos el conjunto distribuido de rangos de índices sobre V definidos por los límites a_j, 0<=j<P y que el número de sumas (a_j - a_{j-1}) por cada procesador esté balanceado. El balanceo de procesamiento es una actividad importante que consiste en la distribución del cómputo en los procesadores con el objetivo de minimizar los tiempos de espera. La cantidad de trabajo más grande asignada a cada procesador, de tal manera que minimice los tiempos de espera, queda determinado por el número de procesadores disponibles y el número de elementos en vector de vértices.

Balanceo de carga de procesamiento en estructuras de acceso no ordenado.

Iniciemos con una técnica de balanceo ingenua y válida, en la que el número de elementos es simplemente queda dividida por el número de procesadores. El cociente entero de ésta división nos indica la cantidad de elementos que cada núcleo, al menos, debería procesar. Y el residuo más el cociente, la cantidad de elementos que deberá procesar uno de los procesadores, de tal manera que todos los elementos se procesen. Así pues, si P es el número de procesadores, N número de elementos de la entrada y W la cantidad de elementos asignada a cada procesador es:

$$W = N/P$$

$$W_{LastProcessor} = W + (N \bmod P)$$

Esto efectivamente cubre el procesamiento de todos los elementos de entrada. Esta técnica, aunque es válida no es la más óptima. Ya que el residuo se concentraría en uno de los procesadores y por lo tanto tardará más en procesar dicha carga de trabajo, lo que aumentará el tiempo de espera requerido para formar la solución total.

En sistemas de cuatro procesadores, este residuo es máximo tres, lo que aparentemente no afecta al rendimiento, pero si tenemos 400 procesadores, el residuo es máximo 399. Lo cual puede impactar de manera considerable. ¿No sería más práctico, distribuir 399 elementos sobrantes a los demás procesadores y quedarnos con un solo procesador que procese un elemento menos? Para

valorar la relevancia de éste problema pensemos en lo siguiente. Por ejemplo, suponga que tenemos 8 núcleos de procesamiento y un volumen de datos de 407 elementos. Utilizando la técnica ingenua de distribución de trabajo, nos quedarían siete procesadores trabajando sobre 50 elementos y un último procesador con 57 elementos, cubriéndose así el procesamiento de la totalidad de los elementos de entrada. El último procesador, tardará 7 instantes más de procesamiento, por lo que la solución total para que sea válida, estable y consistente, podrá producirse dentro de 57 instantes de tiempo, una vez iniciadas las ocho tareas. El problema también tiene otra perspectiva, ya que siete procesadores estarán inutilizados durante 7 instantes de de procesamiento, lo que implica la pérdida de tiempo equivalente al procesamiento de 49 elementos. ¡Qué desperdicio! ¿No?

El rendimiento de ésta técnica puede calcularse y nos servirá como indicador de eficiencia de paralelismo y compararla con otras técnicas. La taza rendimiento paralelo está dado por el número de elementos de entrada N, la asignación de trabajo máximo W_{max} (número de elementos máximo asignados para su procesamiento en cualquiera de los núcleos) y el número de procesadores disponibles P de tal manera que:

$$R = \frac{N}{PW_{max}}$$

Suponiendo los datos del ejemplo anterior y realizando aritmética donde N=407 y P=8 y W_{max}=57, se tiene un rendimiento de procesamiento paralelo de apenas un 89.25%. Equivalente a un pérdida de tiempo del 10.75% no aprovechado en paralelismo. ¿Le resulta ahora importante el balanceo de carga de sus algoritmos paralelos?

Ahora estudiaremos una estrategia óptima de balanceo que maximiza la tasa de rendimiento paralelo R. La técnica consiste en distribuir el residuo de la técnica anterior en los procesadores disponibles, reduciendo el tiempo de espera general.

En términos generales, si conocemos el tamaño de la entrada de datos N, el número de procesadores P y tenemos un algoritmo de acceso no ordenado sobre la entrada, entonces:

$$W_{max} = \left\lfloor \frac{(N + (P-1))}{P} \right\rfloor$$

Donde la carga en cada procesador W_j

$$W_j = W_{max} - \begin{cases} 1, & 0 < (N \bmod P)\ , j \geq (N \bmod P) \\ 0, & En\ cualquier\ otro\ caso \end{cases}$$

De tal manera que se cumple que:

$$N = \sum_{j=0}^{P-1} W_j$$

y es un balanceo óptimo y completo.

Demostración:

En el caso trivial donde el número de elementos de entrada es divisible exactamente entre el número de procesadores, implicaría que N mod P = 0. Por lo que la carga de cada procesador W_j es simplemente W_{max}. W_{max} a su vez es cociente N/P al calcularse el piso de W_{max}. Por lo tanto, se logra una tasa de paralelismo del 100%, ya que las cargas son iguales.

Para el caso en el que el módulo (N mod P) > 0, se tienen N mod P procesadores con asignaciones de tamaño W_{max}, y P-(N mod P) procesadores con asignaciones de tamaño W_{max}-1, por lo que se cumple que la tasa de rendimiento de paralelismo nunca será menor que R=N/(N+1). Para este caso W_{max} es exactamente igual al cociente entero N/P más uno.

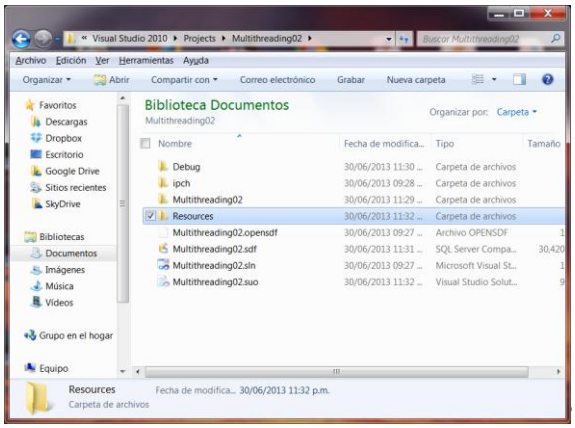
Finalmente, como la diferencia entre las asignaciónes W_j a los procesadores es cero o uno, entonces es el balanceo es el más óptimo posible y el tiempo de espera queda determinado por el lote más grande de dimensión W_{max}.

Realizando un poco de aritmética, y tomando el caso cuando P=8, N=407, y aplicando nuestra nueva estrategia de balanceo, encontraremos que W_{max}=51, con 7 procesadores con carga W_{max} y un procesador con carga 50. Por lo que R=N/PW_{max} nos da una tasa de paralelismo del 99.75%, que es exactamente igual a R=N/(N+1)=407/408. Es evidente que este resultado es muy diferente a la tasa de 89.25% de la estrategia anterior, dando una mejoría del 10.5% en pro del paralelismo. Solo se desperdicia un 0.25% de tiempo de espera y no un 10.75% con la estrategia ingenua. ¡Impresionante!

Implementación del cómputo en paralelo del centroide geométrico de Monkey.

Ya que hemos analizado el problema del cómputo del centroide a partir de la posición de los vértices y estudiado las estrategias de balanceo aplicables, podemos establecer una implementación robusta y eficaz que se adapte a la infraestructura de procesamiento paralelo disponible en cada computador, ya que debemos preparar a la aplicación para que se pueda ejecutar en CPU's con uno o más núcleos de manera óptima. La siguiente implementación no es más que la síntesis de los conceptos anteriores.

Para ello, deberá crear el proyecto Multithreading02, o el nombre que mejor le convenga. Descargue los archivos con la información de los vértices e índices de Monkey. Copie los archivos Monkey.vertex y Monkey.index al directorio "Resources". Si la subdirectorio "Resources" no existe, deberá crearla.



Programa 02

```
// Multithreading02.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <vector>
#include <fstream>
#include <iostream>
#include <Windows.h>
using namespace std;

struct VECTOR4D
{
	float x,y,z,w;
};

bool LoadVertexData(char* pszFileName,vector<VECTOR4D>& vOutVertexData)
{
	OMITIDO POR BREVEDAD
}
bool LoadIndexData(char* pszFileName,vector<unsigned long>& vOutIndexData)
{
	OMITIDO POR BREVEDAD
}

//Estructura de paso de parámetros entre hilos
struct THREADPARAMS
{
	HANDLE	hEventStart;	//Hilo espera orden de inicio
	HANDLE	hEventFinished;	//Hilo notifica su finalización
	//Entrada del procesamiento
	VECTOR4D*		pv4dDataSource;
	unsigned long	ulVertexCount;
	//Salida del procesamiento
	VECTOR4D		v4dPartialSumation;
};
//Prototipo de Hilo : PartialCentroid
DWORD WINAPI ThreadPartialSumation(THREADPARAMS* pParams);


//Hilo primario
int _tmain(int argc, _TCHAR* argv[])
{
	//Consola en español
	SetConsoleOutputCP(1252);
	SetConsoleCP(1252);

	vector<VECTOR4D> MonkeyVertices;
	//Carga de la información de vértices de Monkey
	if(!LoadVertexData("..\\Resources\\Monkey.vertex",MonkeyVertices))
	{
		cout<<"No se pudo abrir el archivo con los vértices de Monkey"<<endl;
		return 0;
	}
	cout<<"Número de vértices cargados:"<<MonkeyVertices.size()<<endl;
	//Detección de infraestructura de procesamiento paralelo "metodo simple"
	unsigned long ulNumberOfLogicalProcessors;
	cout<<"¿Desea autodetectar el número de procesadores lógicos?(s/n)"<<endl;
	{
		char resp;
		cin>>resp;
		if(resp=='s' || resp=='S')
			ulNumberOfLogicalProcessors=atoi(getenv("NUMBER_OF_PROCESSORS"));
		else
		{
			cout<<"Indique el número de procesadores lógicos"<<endl;
			cin>>ulNumberOfLogicalProcessors;
			ulNumberOfLogicalProcessors=max(1,ulNumberOfLogicalProcessors);
		}
		cin.ignore();
	}

	cout<<"Número de procesadores lógicos:"<<ulNumberOfLogicalProcessors<<endl;

	//Crear los objetos de sincronización
	//Crear eventos, de tipo de reinicio manual, con estado inicial no señalizado (false)
```

```
	HANDLE* pahEventFinished=new HANDLE[ulNumberOfLogicalProcessors];
	for(unsigned long ulLogicalProcessor=0;
		ulLogicalProcessor<ulNumberOfLogicalProcessors;
		ulLogicalProcessor++)
	{
		pahEventFinished[ulLogicalProcessor]=CreateEvent(NULL,TRUE,FALSE,NULL);
	}
	HANDLE hEventStart=CreateEvent(NULL,TRUE,FALSE,NULL);


	//Crear hilos y distribución de datos balaceado
	THREADPARAMS *paThreadParams=new THREADPARAMS[ulNumberOfLogicalProcessors];
	THREADPARAMS *pParams=paThreadParams; //Iterador
	unsigned int Wmax=(MonkeyVertices.size()+(ulNumberOfLogicalProcessors-1))/
							ulNumberOfLogicalProcessors;
	unsigned long ulBaseVertex=0;
	unsigned long ulNmodP=MonkeyVertices.size()%ulNumberOfLogicalProcessors;
	for(unsigned long ulLogicalProcessor=0;
		ulLogicalProcessor<ulNumberOfLogicalProcessors;
		ulLogicalProcessor++)
	{
		pParams->hEventFinished=pahEventFinished[ulLogicalProcessor];//Hilo notifica su finalización con
este evento
		pParams->hEventStart=hEventStart;			//Hilo espera ordenes de inicio con este
evento
		pParams->pv4dDataSource=&MonkeyVertices[ulBaseVertex];
		pParams->ulVertexCount=Wmax-((0<ulNmodP)?((ulLogicalProcessor>=ulNmodP)?1:0):0);
		CreateThread(NULL,4096,(LPTHREAD_START_ROUTINE)ThreadPartialSumation,pParams,NULL,NULL);
		ulBaseVertex+=pParams->ulVertexCount;
		cout<<"Hilo "<<ulLogicalProcessor<<" va a procesar "<<pParams->ulVertexCount<<" vértices"<<endl;
		pParams++;
	}
	cout<<"Balanceo de datos y creación de hilos compleatada"<<endl;

	cout<<"Iniciando el cómputo de centroides parciales"<<endl;
	SetEvent(hEventStart);

	//Factor del inverso de N para el computo del centroide.
	//Aprovechando que otros hilos están trabajando.
	float fInvN=1.0f/MonkeyVertices.size();

	//Esperar a que los hilos secundarios terminen su trabajo
	cout<<"Hilo primario se encuentra esperando a que los hilos secundarios terminen\n";
	WaitForMultipleObjects(ulNumberOfLogicalProcessors, pahEventFinished,TRUE,INFINITE);
	cout<<"Las tareas secundarias han finalizado exitosamente."<<endl;
	//Realizar la mezcla de los resultados parciales
	cout<<"Produciendo resultado total"<<endl;
	VECTOR4D v4dTotalSumation={0.0f,0.0f,0.0f,0.0f};
	for(unsigned long ulThread=0;ulThread<ulNumberOfLogicalProcessors;ulThread++)
	{
		v4dTotalSumation.x+=paThreadParams[ulThread].v4dPartialSumation.x;
		v4dTotalSumation.y+=paThreadParams[ulThread].v4dPartialSumation.y;
		v4dTotalSumation.z+=paThreadParams[ulThread].v4dPartialSumation.z;
		v4dTotalSumation.w+=paThreadParams[ulThread].v4dPartialSumation.w;
	}
	v4dTotalSumation.x*=fInvN;
	v4dTotalSumation.y*=fInvN;
	v4dTotalSumation.z*=fInvN;
	v4dTotalSumation.w*=fInvN;
	cout<<"El centroide es:["<<
		v4dTotalSumation.x<<", "<<v4dTotalSumation.y<<", "<<
		v4dTotalSumation.z<<", "<<v4dTotalSumation.w<<"]"<<endl;
	//Desalojar recursos de procesamiento
	delete [] paThreadParams;
	//Desalojar recursos de sincronización
	for(unsigned long ulLogicalProcessor=0;
		ulLogicalProcessor<ulNumberOfLogicalProcessors;
		ulLogicalProcessor++)
	{
		CloseHandle(pahEventFinished[ulLogicalProcessor]);
	}
	delete [] pahEventFinished;
	CloseHandle(hEventStart);
	return 0;
}

//Implementación de PartialCentroid
DWORD WINAPI ThreadPartialSumation(THREADPARAMS* pParams)
{
```

```
        //Esperar la orden para comenzar
        WaitForSingleObject(pParams->hEventStart,INFINITE);
        //Extraer una copia del evento hEventFinished antes de liberar la estructura de parámetros

        //Sumatoria de los vértices asignados
        VECTOR4D v4dSumation={0.0f,0.0f,0.0f,0.0f};
        VECTOR4D* pv4dV=pParams->pv4dDataSource;
        for(int a=0;a<pParams->ulVertexCount;a++)
        {
                v4dSumation.x+=pv4dV->x;
                v4dSumation.y+=pv4dV->y;
                v4dSumation.z+=pv4dV->z;
                v4dSumation.w+=pv4dV->w;
                pv4dV++;
        }
        pParams->v4dPartialSumation=v4dSumation;
        //Notificar que éste hilo ha terminado
        SetEvent(pParams->hEventFinished);
        return 0;
}
```

Actividades:

- Implemente el programa anterior, analícelo, compréndalo y experimente.
- Modifique el programa para calcular el área superficial. Por lo que será necesario cargar el vector de índices. El área superficial, no es más que la sumatoria de las áreas correspondientes a cada uno de los triángulos de la malla.
- Publique su resultado en el grupo de Facebook.
- Realice preguntas en caso de dudas acerca de ésta actividad en el grupo de Facebook.

Sincronización de hilos y la exclusión mutua

Cuando compartimos una variable o estructura de datos entre dos o más hilos, es importante implementar un modelo de consistencia de memoria, los cuales, permiten que un algoritmo paralelo obtener los mismos resultados tal y como se obtienen mediante un algoritmo secuencial que resuelve el mismo problema.

La memoria RAM y los periféricos del computador, son recursos compartidos entre los varios procesadores que se tengan disponibles, y por lo tanto, el acceso a estos recursos es no ordenado, ya que cualquier procesador o tarea específica, puede lograr el acceso a estos recursos en cualquier momento. Si este acceso no es "arbitrado" o controlado, la inconsistencia es la consecuencia.

La *exclusión mutua*, es una propiedad que permite que una y sólo una tarea a la vez modifique u opere sobre un recurso, sea memoria o periférico.

En el caso de una variable, supongamos que dos hilos A y B tienen acceso a una variable X inicialmente con el valor 0. Si los hilos A y B leen X, incrementan su valor y lo escriben de nuevo en X, es decir, cada hilo ejecuta la secuencia {t=X; t++; X=t;}, entonces ¿cuál es el valor final de la variable X? Pueden suceder varias situaciones y por supuesto esto ya es inconsistencia. Supongamos que el hilo A, ejecuta todas sus instrucciones antes que B inicie. Entonces, el valor final de X es 2. Pero si A y B leen la variable X, incrementan y luego almacenan, ambos hilos A y B escribirían un 1 en X. Esto es inconsistencia debido al paralelismo de ejecución y a la duplicidad del valor de X durante estas ejecuciones, ya que tenemos a un kernel que solo planifica y ejecuta hilos.

Si aplicamos un modelo de consistencia de datos, el programa secuencial que resuelve este problema de forma determinista, sería la secuencia {t=X; t++; X=t;} y luego {t=X; t++; X=t;}. Así el valor final de X tras ejecutar estas secuencias sería siempre 2.

Para evitar el problema de duplicidad del estado de una variable (valor), es necesario establecer que el acceso y operaciones sobre su estado sea una *sección crítica*. Para nuestro ejemplo, una sección crítica implementa *exclusión mutua* entre varios hilos sobre el estado de la variable X. La sección crítica, para nuestro ejemplo, es la secuencia {t=X; t++; X=t;} lo que implica que se tiene acceso exclusivo a X, durante y solo ese periodo de ejecución.

Sincronización de hilos mediante Sección Crítica

Una *sección crítica*, es un conjunto C_j de secuencias finitas de instrucciones S_i={*Secuencia de instrucciones*} que deben ejecutarse de manera atómica, de tal manera que solo exista uno y sólo un contexto de tarea o hilo ejecutando una y solo una de esas secuencias de C_j. Es claro, que cualquier hilo que no esté en ninguna de esas secuencias deben continuar con su ejecución planificada.

Al solo permitir que una sola tarea se encuentre ejecutando la sección crítica, es evidente una desaceleración del paralelismo, por lo que es necesario reducir la frecuencia de entrada a

secciones críticas y su ejecución debe ser breve si no deseamos afectar al rendimiento general de nuestra solución.

Nota: Las secciones críticas nos permiten implementar acceso de exclusión mutua a una colección de variables o dispositivos, pero no es la única forma de llevarla a cabo. Existen otras formas que garantizan la exclusión mutua sin la necesidad de que varios hilos esperen demasiado, por ejemplo, las líneas de trabajo o "pipelines", en la que cada hilo está ejecutando tareas diferentes con su propio estado en el mismo instante, tal y como sucede en las líneas de producción de manufactura.

El sistema operativo provee objetos de sincronización como la sección crítica para indicar al planificador de tareas cuando continuar o detener la ejecución de los hilos que están a punto de ejecutar la sección crítica. La aplicación y las tareas, deben cooperar con el sistema operativo para indicar cuando éstas entran y salen de la sección crítica, así el kernel, apoyará con la ejecución o suspensión de las tareas.

Para lograr la ejecución atómica de instrucciones S_i bajo la sección crítica C_j, se debe respetar la siguiente secuencia o traza de instrucciones { Entrar(C_j); S_i ; Salir(C_j) }. En donde Enter(C_j) , es una función que solicita ejecución de la sección critica de instrucciones bajo el control de C_j al sistema operativo. Si hay una secuencia S_i en ejecución bajo el control de C_j, el hilo que solicita la sección crítica, esperará a que otro hilo libere la sección crítica mediante la función Exit(C_j). Los hilos que esperan entran en suspensión hasta que la sección crítica quede liberada. Si hay varios hilos esperando la sección crítica, por propiedad de justicia, serán atendidos y entrarán a la sección crítica según el turno de la solicitud (First In, First Out).

El siguiente mapa de memoria de código muestra el concepto de sección crítica aplicado.

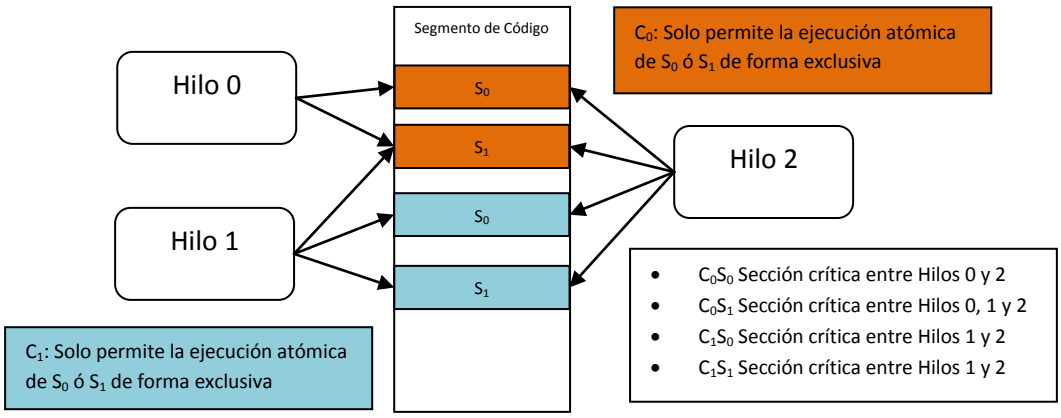


Ilustración 7 Una posible asignación y distribución de secciones críticas entre hilos

La Ilustración 7 nos muestra indica qué regiones de código o secuencias de instrucciones deberán ser ejecutadas de manera atómica entre los hilos involucrados. Por ejemplo, si el Hilo 0 logró entrar a la sección crítica C_0S_1, implica que el Hilo 1 no podrá ejecutar esa sección crítica hasta que

el Hilo 0 la libere, pero el Hilo 1 puede lograr acceso a otras secciones criticas bajo C_1 si están disponibles. Ahora bien, si el Hilo 0 entró a cualquier sección crítica bajo C_0 y el Hilo 1 entró a cualquier sección crítica bajo C_1, entonces el Hilo 2 deberá esperar a que se liberen o C_1 o C_2. Un hilo solo puede esperar a una liberación de sección crítica ya que queda bloqueado durante ese periodo de espera.

Implementación de un servicio de mensajería instantánea entre varios clientes y un servidor: Comunicación entre procesos y máquinas

Para mostrar el uso de las secciones críticas y la comunicación entre procesos y máquinas, vamos a implementar un pequeño servicio de mensajería instantánea. Para que los clientes puedan comunicarse entre sí, podemos hacer uso de un programa servidor que distribuya los mensajes entre los clientes.

Los clientes se conectan al servidor que provee el sistema de mensajería, y el servidor atiende las solicitudes de los clientes para enviar sus mensajes a todos los demás clientes. El cliente solo emplea un hilo secundario para recibir los mensajes de otros clientes, mientras que el hilo primario se dedica a leer los mensajes del teclado y los envía al servidor.

El servidor debe administrar una colección de conexiones y sesiones abiertas. Cada sesión abierta es procesada por un par de hilos, uno de ellos se encarga de recibir mensajes de su cliente asignado y el otro de revisar su buzón de salida para enviarle los mensajes provenientes de otros clientes. El hilo receptor de mensajes, coloca el mensaje en los buzones de salida de las otras sesiones abiertas. Las sesiones son registros que agrupan la clave de sesión, el socket de conexión con el cliente y el mismo buzón de salida hacia un cliente.

Es claro que el acceso al mapa de sesiones necesita acceso exclusivo, y es por ello que utilizaremos una sección crítica para que cuando la tabla de sesiones sea accedida, no se presenten inconsistencias que pueden dañar el estado del servicio.

Implementación del cliente de mensajería

Comenzaremos a analizar el programa más simple, el cliente. A simple, nos referimos a la simplicidad de las actividades que el cliente debe realizar desde el punto de vista del servicio de mensajería. Recibe y envía mensajes al servidor. El siguiente código define el comportamiento del servidor.

El cliente inicia pidiendo el nombre del servidor o la dirección IP del mismo para realizar una conexión con el servicio de mensajería.

Para mayor información sobre el uso de sockets de internet, le recomiendo lea el tutorial de sockets de internet publicado en el grupo de Facebook.

Programa 03 – Cliente de Mensajería

```
// Client.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#pragma comment(lib,"WS2_32.lib")
#include <WinSock2.h>
#include <Windows.h>
#include <iostream>
#include <string>
using namespace std;

struct THREAD_CLIENT_PARAMS
{
	//Datos del servidor y servicio (puerto)
	SOCKADDR_IN service;
	//Socket con el servidor y el servicio
	SOCKET sktService;
};
DWORD WINAPI ThreadClientReceive(THREAD_CLIENT_PARAMS *pParams);

int _tmain(int argc, _TCHAR* argv[])
{
	//Consola en español
	SetConsoleOutputCP(1252);
	SetConsoleCP(1252);

	cout<<"Inicializando cliente..."<<endl;
	WSADATA wsadata;
	//Cargar la DLL de Windows Sockets
	if(WSAStartup(MAKEWORD(2,2),&wsadata))
	{
		cout<<"No se pudo iniciar la API de Sockets de Windows v2.2"<<endl;
		return 0;
	}
	//Datos del servidor y su dirección IP, Entidad de Internet
	HOSTENT *pHostEnt;
	//Solicitar IP o nombre del servidor
	string strIPAddress;
	cout<<"Introduzca la dirección IP o nombre del servidor al que desea conectarse:"<<endl;
	cin>>strIPAddress;
	cin.ignore();
	//Resolver la dirección del servidor a partir de su nombre o cadena IP.
	pHostEnt=gethostbyname(strIPAddress.c_str());
	if(!pHostEnt)
	{
		cout<<"No se pudo resolver el nombre del servidor"<<endl;
		return 0;
	}
	//Se ha resuelto el nombre del servidor, por que podemos intentar realizar una conexión
	SOCKADDR_IN server;
	memset(&server,0,sizeof(SOCKADDR_IN));
	server.sin_addr=*(IN_ADDR*)pHostEnt->h_addr;	//Dirección resuelta del servidor
	server.sin_port=htons(6120);			//Puerto de escucha (servicio)
	server.sin_family=AF_INET;			//Familia de protocolos de internet

	//Crear un socket TCP
	SOCKET sktService=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
	//Conectar con el servicio de mensajería
	cout<<"Conectando con:"<<inet_ntoa(server.sin_addr)<<endl;
	if(-1==connect(sktService,(SOCKADDR*)&server,sizeof(server)))
	{
		cout<<"El servicio no está disponible en estos momentos, inténtelo más tarde"<<endl;
		return 0;
```

```
        }
        cout<<"Conexión establecida con:"<<inet_ntoa(server.sin_addr)<<endl;
        //Se logrado conexión con el servidor....
        cout<<"Iniciando tareas de cliente..."<<endl;


        //Crear el hilo receptor de mensajes provenientes del servicio de mensajería
        THREAD_CLIENT_PARAMS *pParams=new THREAD_CLIENT_PARAMS;
        pParams->service=server;
        pParams->sktService=sktService;
        CreateThread(NULL,8192,(LPTHREAD_START_ROUTINE)ThreadClientReceive,pParams,0,NULL);

        cout<<"Escriba su mensaje y presione <ENTER> cada vez que desee enviar uno."<<endl;
        while(1)
        {
                //Esperar a que el usuario introduzca un mensaje
                char szMessage[1024];
                memset(szMessage,0,sizeof(szMessage));
                cin.getline(szMessage,1024);
                string strMessage;
                strMessage=szMessage;
                //Enviar la longitud en bytes del mensaje
                long lMessageLength=strMessage.length();
                int nRetVal=
                send(sktService,(char*)&lMessageLength,sizeof(long),0);
                if(-1==nRetVal)
                {
                        cout<<"El servicio ya no está disponible..."<<endl;
                        break;
                }
                nRetVal=send(sktService,strMessage.c_str(),lMessageLength,0);
                if(-1==nRetVal)
                {
                        cout<<"El servicio ya no está disponible..."<<endl;
                        break;
                }
        }
}

//Hilo secundario de recepción de mensajes desde el servidor
DWORD WINAPI ThreadClientReceive(THREAD_CLIENT_PARAMS *pParams)
{
        while(1)
        {
                //Esperar a la recepción de un mensaje
                char* pszBuffer;
                long lMessageLength;
                int nRetVal;
                nRetVal=recv(pParams->sktService,(char*)&lMessageLength,sizeof(long),0);
                if(-1==nRetVal)
                {
                        cout<<"El servicio ya no está disponible..."<<endl;
                        break;
                }
                if(0==nRetVal)
                {
                        cout<<"El servidor ha cerrado la conexión"<<endl;
                        break;
                }
                //Asignar recursos para recibir el mensaje
                pszBuffer=new char[lMessageLength+1];
                pszBuffer[lMessageLength]=0; //Terminar la cadena con nulo
                //Recibir el contenido del mensaje
                nRetVal=recv(pParams->sktService,pszBuffer,lMessageLength,0);
                if(-1==nRetVal)
                {
                        delete [] pszBuffer;
                        cout<<"El servicio ya no está disponible..."<<endl;
                        break;
```

```
                }
                if(0==nRetVal)
                {
                        delete [] pszBuffer;
                        cout<<"El servidor ha cerrado la conexión"<<endl;
                        break;
                }
                //Imprimir el mensaje en consola
                cout<<endl<<pszBuffer<<endl;
                delete [] pszBuffer;
        }
        delete pParams;
        return 0;
}
```

Implementación del servidor de mensajería

El servidor es más complejo que la funcionalidad del cliente anterior. Esto se debe a que el servidor debe administrar las conexiones con múltiples clientes, en contraste con el hecho de que un cliente solo debe operar con un servidor.

El servidor, debe recibir las conexiones de los clientes que deseen conectarse con él, por lo que una de las tareas que debe hacer el servidor, es atender estas solicitudes de conexión. Para ello delega esta actividad a un hilo secundario que escucha solicitudes de conexión. Este hilo, es el encargado de aceptar las conexiones entrantes y crear las sesiones para atender a cada cliente. Una vez creada y registrada la sesión, se crean dos hilos secundarios adicionales que atienden envío y recepción de mensajes por cliente aceptado.

Cuando el hilo de recepción de mensajes, capta un mensaje desde su cliente, éste inmediatamente hace la difusión de los mensajes en los buzones de salida de otras sesiones.

Cuando el hilo de envío de mensajes, detecta que su buzón de salida no está vacío, entonces transmite ese mensaje a su cliente y elimina el mensaje del buzón de salida. El hilo de envío de mensajes revisa, de manera periódica cada 100ms, al buzón de salida y verifica si es necesario enviar otro mensaje al cliente.

Cada par de hilos de envío y recepción de mensajes está asignado a una sesión y por lo tanto atienden un cliente. Es por ello que el servidor creará otro par de hilos de servicio por cada cliente conectado. Los hilos se destruyen automáticamente, cuando el cliente abandona la conexión o la cierra.

Como podrá verificar en el siguiente programa, todos los hilos recurren a la colección de sesiones por lo que será necesario implementar una estrategia de exclusión mutua para evitar que varios hilos modifiquen dicha colección simultáneamente y evitar que otros hilos lean información corrupta debido a la concurrencia.

Se han marcado en el código a las secciones críticas que vigilan el acceso de exclusión mutua sobre el mapa de sesiones.

Programa 04 – Servidor de Mensajería

```
// Server.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#pragma comment(lib,"WS2_32.lib")
#include <WinSock2.h>
#include <Windows.h>
#include <iostream>
#include <string>
#include <map>
#include <list>
using namespace std;

struct MESSAGE
{
	//Un mensaje de texto
	string strContents;
	//Agregar más información de mensaje aquí

};
struct SESSION
{
	//Clave única de sesión
	long lSessionID;
	//Buzón de salida hacia un cliente
	list<MESSAGE> lstOutBox;
	//Agregar más informacion de sesión aquí
};

struct LISTENER_THREAD_PARAMS
{
	//Dirección de éste servidor
	sockaddr_in server;
	//Socket de escucha de conexiones entrantes en el servidor
	SOCKET	sktListener;
};

struct SERVICE_THREAD_PARAMS
{
	//Contador de referencias, indica cuantos hilos están usando este socket y ésta sesión
	long		lRefCount; //Cuando el contador llegue a cero, la sesión será eliminada
	//Clave de la sesión en el mapa de sesiones
	long		lSessionID;
	//Dirección del cliente
	SOCKADDR_IN client;
	//Socket de conexión con el cliente
	SOCKET		sktClient;
};


//Sección critica que protege con exclusión mutua a los métodos de g_mapSessions y el acceso a g_lSessionGenerator
CRITICAL_SECTION g_csSessions;
//El mapa de sesiones
map<long,SESSION*> g_mapSessions;
//Generador de claves de sesión
long g_lSessionGenerator;

//Hilos secundarios

//Hilo de envío de mensajes pendientes en el buzón de salida de un cliente
DWORD WINAPI ServiceThreadSend(SERVICE_THREAD_PARAMS* pParams);

//Hilo de recepción de mensajes de un cliente y distribución en buzones de salida de otras sesiones
DWORD WINAPI ServiceThreadReceive(SERVICE_THREAD_PARAMS* pParams);

//Hilo receptor de solicitudes de conexión con éste servicio
DWORD WINAPI ListenerThread(LISTENER_THREAD_PARAMS* pParams);


//Hilo primario
int _tmain(int argc, _TCHAR* argv[])
{
	//Consola en español
	SetConsoleOutputCP(1252);
	SetConsoleCP(1252);
	cout<<"Inicializando servidor..."<<endl;
```

```
	WSADATA wsadata;
	//Cargar la DLL de Windows Sockets
	if(WSAStartup(MAKEWORD(2,2),&wsadata))
	{
		cout<<"No se pudo iniciar la API de Sockets de Windows v2.2"<<endl;
		return 0;
	}
	cout<<"Preparando socket de escucha de conexiones..."<<endl;
	//El servidor crea un socket de escucha
	SOCKADDR_IN server;
	memset(&server,0,sizeof(SOCKADDR_IN));
	server.sin_addr.s_addr=INADDR_ANY;   //Asocia el socket con la ip del equipo
	server.sin_port=htons(6120);         //El puerto de escucha uno de 1024 a 65535
	server.sin_family=AF_INET;           //Internet protocol

	cout<<"Creando el socket de escucha TCP/IP..."<<endl;
	SOCKET sktListener;
	sktListener=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
	if(-1==sktListener)
	{
		cout<<"No se pudo crear el socket de escucha."<<endl;
		return 0;
	}
	cout<<"Socket de escucha creado exitosamente."<<endl;
	//Atar el socket con la dirección y puerto IP
	cout<<"Levantando servicio en el puerto:"<<htons(server.sin_port)<<" ..."<<endl;
	if(-1==bind(sktListener,(sockaddr*)&server,sizeof(sockaddr_in)))
	{
		cout<<"No se pudo atar el socket al puerto especificado, es posible que el puerto"<<
			htons(server.sin_port)<<
			" ya esté en uso por otro servicio en éste servidor."<<endl;
		return 0;
	}
	cout<<"Asociación del puerto de escucha con ésta aplicación completada."<<endl;
	//Iniciar el hilo secundario de escucha y aceptación de conexiones con clientes
	cout<<"Comenzando tareas de servidor..."<<endl;
	//Crear objetos de sincronización
	InitializeCriticalSection(&g_csSessions);
	//El hilo secundario ListenerThread, recibirá nuevas conexiones,
	//creará las nuevas sesiones y sus hilos de servicio
	LISTENER_THREAD_PARAMS *pParams=new LISTENER_THREAD_PARAMS;
	pParams->server=server;
	pParams->sktListener=sktListener;
	CreateThread(NULL,32*1024,(LPTHREAD_START_ROUTINE)ListenerThread,pParams,0,NULL);
	cout<<"¡Servidor listo y motores en funcionamiento!"<<endl;
	cout<<"Para terminar con el servidor, presione <ENTER> en cualquier momento."<<endl;
	cin.ignore();
	return 0;
}

//Función que libera los recursos de sesión, hasta que ningun otro hilo la necesite.
void FreeServiceThreadResources(SERVICE_THREAD_PARAMS* pParams)
{
	EnterCriticalSection(&g_csSessions);
	//Si al decrementar atómicamente al contador de referencias a los parametros llega a cero:
	if(0 == --pParams->lRefCount)
	{
		//No hay más hilos utilizando esta información, proceder con la eliminación
		//de los recursos de la sesión.
		closesocket(pParams->sktClient);
		//Eliminar la sesión del mapa de sesiones
		auto it=g_mapSessions.find(pParams->lSessionID);
		delete it->second;
		g_mapSessions.erase(it);
		//Finalmente liberar la memoria de parámetros de servicios
		delete pParams;
	}
	LeaveCriticalSection(&g_csSessions);
}

DWORD WINAPI ServiceThreadSend(SERVICE_THREAD_PARAMS* pParams)
{
	while(1)
	{
		//Revisar el buzón de salida hacia éste cliente

		//Solicitar acceso exclusivo al contenido de las sesiones
		EnterCriticalSection(&g_csSessions);
		auto itSession=g_mapSessions.find(pParams->lSessionID);
```

```
                SESSION *pSession=itSession->second;
                //Mientras existan mensajes de salida pendientes...
                if(!pSession->lstOutBox.empty())
                {
                        //Extraer el mensaje más antiguo recibido en este buzón
                        MESSAGE msg;
                        msg=pSession->lstOutBox.front();
                        pSession->lstOutBox.pop_front();
                        //Liberar la sección crítica, ya tenemos una copia del mensaje
                        LeaveCriticalSection(&g_csSessions);
                        //Enviar información al cliente
                        long lMessageLength;
                        lMessageLength=msg.strContents.length();
                        int nRetVal= send(pParams->sktClient,(char*)&lMessageLength,sizeof(long),0);
                        if(-1==nRetVal)
                        {
                                cout<<"Se ha perdido conexión con el cliente..."<<endl;
                                break; //Salir del ciclo de envío y salir de éste hilo
                        }
                        //Enviar el mensaje
                        nRetVal= send(pParams->sktClient,msg.strContents.c_str(),lMessageLength,0);
                        if(-1==nRetVal)
                        {
                                cout<<"Se ha perdido conexión con el cliente..."<<endl;
                                break; //Salir del ciclo de envío y salir de éste hilo
                        }
                }
                else //No hay mensajes pendientes, liberar la sección critica
                {
                        //Liberar el acceso exclusivo a los datos de las sesiones
                        LeaveCriticalSection(&g_csSessions);
                }
                Sleep(100); //Revisar el buzón de salida dentro de 100ms para ahorrar energía
        }
        shutdown(pParams->sktClient,SD_SEND);
        FreeServiceThreadResources(pParams);
        return 0;
}
DWORD WINAPI ServiceThreadReceive(SERVICE_THREAD_PARAMS* pParams)
{
        while(1)
        {
                int nMessageLength=0;
                //Recibir y esperar la longitud del mensaje del cliente. La función returna en caso de
                //error, desconexión o buffer completado
                int nRetVal=recv(pParams->sktClient,(char*)&nMessageLength,sizeof(long),0);
                if(0==nRetVal)
                {
                        cout<<"La conexión con:"<<inet_ntoa(pParams->client.sin_addr)<<" ha terminado."<<endl;
                        break;
                }
                else if(-1==nRetVal)
                {
                        cout<<"Ha ocurrido un error con la conexión con:"<<
                                inet_ntoa(pParams->client.sin_addr)<<endl;
                        break;
                }
                //Recibir y esperar el mensaje (contenido) del cliente. La función retorna en caso de
                //error, desconexión o buffer completado
                char* pszBuffer=new char[nMessageLength+1];
                pszBuffer[nMessageLength]=0; //Terminar el buffer con 0
                nRetVal=recv(pParams->sktClient,pszBuffer,nMessageLength,0);
                if(0==nRetVal)
                {
                        cout<<"La conexión con:"<<inet_ntoa(pParams->client.sin_addr)<<" ha terminado."<<endl;
                        delete [] pszBuffer;
                        break;
                }
                else if(-1==nRetVal)
                {
                        cout<<"Ha ocurrido un error con la conexión con:"<<inet_ntoa(pParams->client.sin_addr);
                        delete [] pszBuffer;
                        break;
                }

                //Ya que hemos recibido el mensaje, debemos hacer difusión del mensaje
                //con los demás clientes conectados
                EnterCriticalSection(&g_csSessions);
                for(auto it=g_mapSessions.begin();it!=g_mapSessions.end();it++)
```

```
                {
                        if(it->first!=pParams->lSessionID) //No enviar el mensaje a si mismo
                        {
                                MESSAGE msg;
                                msg.strContents=pszBuffer;
                                it->second->lstOutBox.push_back(msg);
                        }
                }
                //Ya que se hizo la difusión del mensaje, abandonar la sección critica sobre las sesiones.
                LeaveCriticalSection(&g_csSessions);
                delete [] pszBuffer;
        }
        //Cerrar conexión de entrada
        shutdown(pParams->sktClient,SD_RECEIVE);
        //Liberar los recursos de manera sincrónica y segura
        FreeServiceThreadResources(pParams);
        return 0;
}
DWORD WINAPI ListenerThread(LISTENER_THREAD_PARAMS* pParams)
{
        //Hacer el rol principal del servidor
        while(1)
        {
                SOCKET sktClient;
                //Escuchar el socket, en el puerto establecido previamente para recibir clientes
                if(-1==listen(pParams->sktListener,SOMAXCONN))
                {
                        cout<<"Error al intentar escuchar. El servidor terminará."<<endl;
                        break;
                }
                SOCKADDR_IN client;
                int addrsize=sizeof(SOCKADDR_IN);
                sktClient=accept(pParams->sktListener,(SOCKADDR*)&client,&addrsize);
                if(-1==sktClient)
                {
                        cout<<"Hay un problema al intentar aceptar la conexión con un cliente."<<endl;
                }
                else //La conexión ha sido aceptada
                {
                                //Parametrizar los hilos de servicio
                                SERVICE_THREAD_PARAMS* pServiceParams=new SERVICE_THREAD_PARAMS;
                                pServiceParams->client=client;
                                pServiceParams->sktClient=sktClient;
                                pServiceParams->lRefCount=2; //Dos hilos obtendrán estos parámetros.

                                //Entrar a la sección critica, g_csSessions,
                                //y lograr acceso exclusivo a los datos de sesión
                                EnterCriticalSection(&g_csSessions);
                                SESSION* pSession=new SESSION();
                                pSession->lSessionID=g_lSessionGenerator++;
                                g_mapSessions.insert(make_pair(pSession->lSessionID,pSession));
                                pServiceParams->lSessionID=pSession->lSessionID;
                                //Salir de la sección crítica
                                LeaveCriticalSection(&g_csSessions);
                                //Crear el hilo de servicio a cliente de envío de mensajes
                CreateThread(NULL,32*1024,(LPTHREAD_START_ROUTINE)ServiceThreadSend,pServiceParams,0,NULL);
                                //Crear el hilo de servicio a cliente de recepción de mensajes
                CreateThread(NULL,32*1024,(LPTHREAD_START_ROUTINE)ServiceThreadReceive,pServiceParams,0,NULL);
                }
        }
        cout<<"El servicio ya no está atendiendo más solicitudes entrantes."<<endl;
        delete pParams;
        return 0;
}
```

Actividades:

- Implemente los programas anteriores, analícelos, compréndalos y experimente.
- Modifique el programa que permita conocer quien envía el mensaje mediante un sobrenombre (nickname). El cliente solicita el nickname antes de conectarse.
- Realice preguntas en caso de dudas acerca de ésta actividad en el grupo de Facebook.

Arquitectura de GPU

Los Unidades de Procesamiento Gráfico son coprocesadores especializados en operaciones gráficas.