PROGRAMACIÓN SOBRE REDES

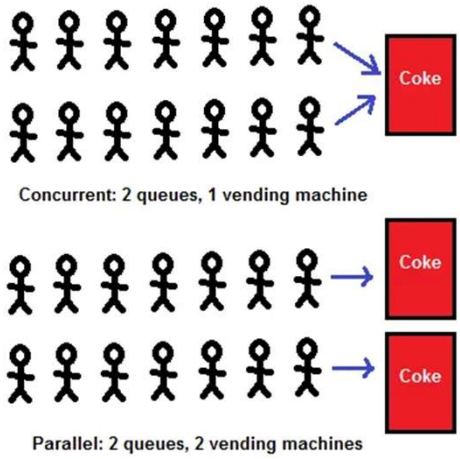
Año 2023
2do Cuatrimestre

Profesor: Victor D'Angela
victor.dangela@bue.edu.ar

Evaluación	2x Evaluación parcial
	1x Proyecto grupal Grupos por definir Defensa individual
	70% asistencia
	Nota mínima de 7 en todas las evaluaciones para promocionar
Clases	Remota a través de Google Meet. Excepto parciales.
	Lunes: contenido nuevo
	Martes: aclaración de dudas previa agenda
	Grupo de Telegram para consultas.

- 14/08/2023 — Presentación e inicio de clases
- 11/09/2023 — Primer parcial
- 18/09/2023 — Recuperatorio primer parcial
- 30/10/2023 — Segundo parcial
- 06/11/2023 — Recuperatorio segundo parcial y entrega de proyecto
- 13/11/2023 — Defensa proyecto
- 27/11/2023 — Recuperatorio defensa proyecto

PROGRAMACIÓN CONCURRENTE



Antiguamente las computadoras arquitectónicamente hablando fueron creadas siguiendo el diseño lógico denominado Arquitectura de Von Newmann, la cual contaba con una unidad de procesamiento, una unidad de control, entrada y salida; y la unidad de procesamiento y control formaban la unidad de procesamiento central CPU, pero este diseño contaba con una sola unidad de procesamiento por lo que los programas debían ser escritos para funcionar bajo este diseño y eso implicaba escribir código que se ejecutara de manera secuencial.

Con base en el anterior fundamento pensemos en la siguiente premisa, una computadora con una sola CPU es capaz de ejecutar una sola operación a la vez, pero que sucede si esa operación es muy trabajosa y lleva mucho tiempo de ejecución para esa CPU. Mientras se ejecuta ese proceso las demás operaciones quedarían en pausa, obviamente esto significa que toda la computadora se congelaría y no respondería aparentemente.

Entonces, ¿qué es la programación concurrente? La concurrencia es la capacidad de ejecutar varios programas o varias partes de un programa a la vez.

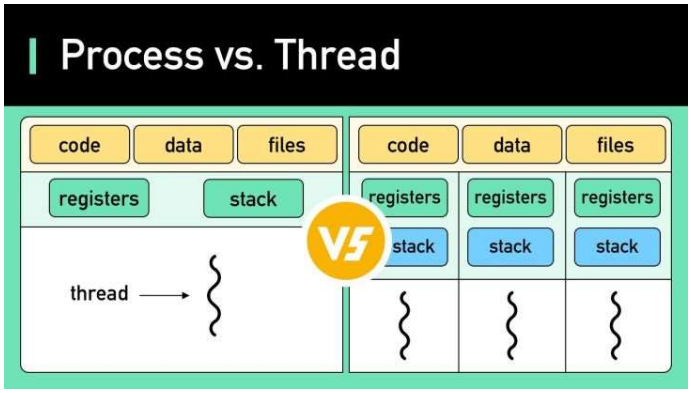
Una computadora moderna tiene varias CPU o varios núcleos dentro de una CPU. La

capacidad de aprovechar estos núcleos múltiples es la clave de muchas aplicaciones. Varios factores llevaron al desarrollo de sistemas operativos que permitió que múltiples programas se ejecutaran simultáneamente:

1. Utilización de recursos: a veces, los programas tienen que esperar operaciones externas. Es más eficiente usar ese tiempo de espera para permitir que se ejecute otro programa.
2. Equidad: varios usuarios y programas pueden tener los mismos derechos sobre los recursos de la máquina. Es preferible permitirles compartir la computadora a través de un intervalo de tiempo más detallado que dejar que un programa se ejecute hasta el final y luego inicie otro.
3. Conveniencia: a menudo es más fácil o más deseable escribir varios programas que realicen una sola tarea y hacer que se coordinen entre sí según sea necesario que escribir un solo programa que realice todas las tareas.

Las mismas preocupaciones (utilización de recursos, equidad y conveniencia) que motivaron el desarrollo de procesos también motivaron el desarrollo de subprocesos (o hilos).

PROCESOS E HILOS: ¿QUÉ SON?



Un proceso es la ejecución de un programa. Incluye el programa en sí, los datos, los recursos, como los archivos, y la información de ejecución, como la información de relación del proceso que mantiene el sistema operativo. El sistema operativo permite a los usuarios crear, programar y terminar los procesos a través de llamadas al sistema.

Un hilo es un semiproceso. Tiene su propia pila y ejecuta una determinada pieza de código. A diferencia de un proceso real, el subproceso normalmente comparte su memoria con otros subprocesos. Por el contrario, los procesos suelen tener un área de memoria diferente para cada uno de ellos.

Algunas características relevantes de los procesos

1. Los procesos no comparten datos e información; son entidades de ejecución aisladas. En resumen, un proceso tiene su propia pila, memoria y datos.

2. Para crear más de un proceso, necesitamos usar llamadas al sistema separadas. Además, se requieren más llamadas al sistema para la gestión de procesos.

3. Finalmente, para cooperar con más de un proceso, necesitamos utilizar mecanismos de comunicación entre procesos (IPC). Esta situación conduce también a un aumento en el número de llamadas al sistema.

Algunas características relevantes de los hilos

1. A diferencia de los procesos, los hilos comparten datos e información. Sin embargo, tienen su propia pila.

2. Podemos crear más de un hilo usando solo una llamada al sistema.

3. Para simplificar aún más las cosas, la gestión de subprocesos requiere pocas o incluso ninguna llamada al sistema porque no necesitamos mecanismos adicionales como IPC para mantener la comunicación entre subprocesos.

PROCESOS E HILOS: DIFERENCIAS

	Proceso	Hilo
Definición	Es un programa en ejecución	Es un semiproceso de un programa
Creación	Requiere mas de una llamada al sistema para crear mas de un proceso	Se pueden crear múltiples hilos con una sola llamada de sistema
Terminación	Toma mas tiempo	Toma menos tiempo
Comunicación	Se requieren mecanismos como IPC para comunicarse entre procesos	No requiere ningún mecanismo especial
Cambio de contexto	Es mas lento	Es mas rápido
Recursos	Pueden consumir mas recursos porque tienen espacios de memoria aislados	Consumen menos recursos
Memoria	Aislada entre procesos	Comparten memoria

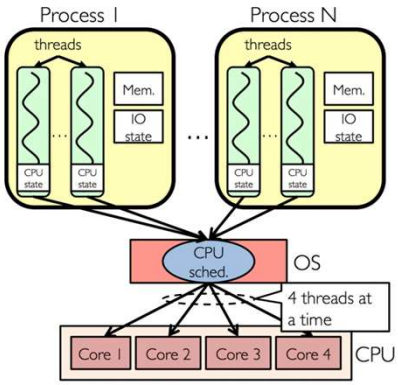
LOS HILOS...

1. … permiten múltiples actividades concurrentes o paralelas dentro de un solo proceso.
2. … son una serie de sentencias ejecutadas.
3. … son una secuencia anidada de llamadas a métodos.
4. …es conocido como proceso ligero o subproceso.
5. …tienen su propio contador de programa, pila y variables locales.

¿POR QUÉ USAR HILOS?

1. Los subprocesos ayudan a realizar procesamiento en segundo plano o asíncrono.
2. Aumenta la capacidad de respuesta de las aplicaciones con GUI.
3. Los subprocesos aprovechan los sistemas multiprocesador.
4. Simplifica la lógica del programa cuando hay varias entidades independientes.*

¿POR QUÉ USAR HILOS?



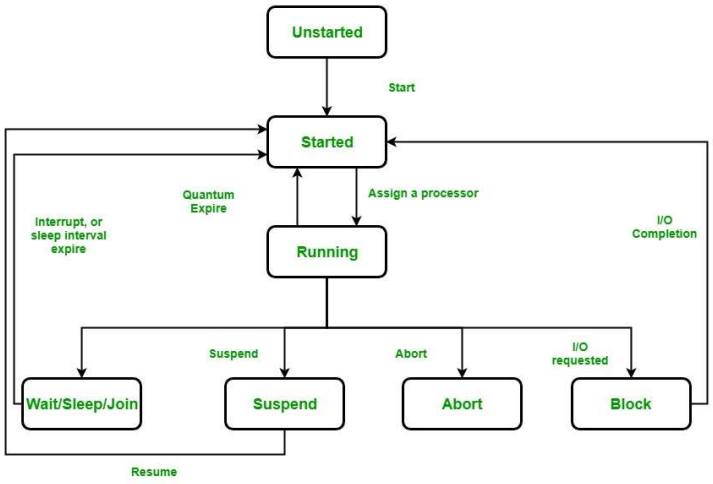
La unidad básica de asignación de tiempo de procesador es el hilo

Dado que la unidad básica de asignación de tiempo de procesador es el hilo, un programa con sólo un subproceso puede ejecutarse como máximo en un procesador a la vez. En un sistema de dos procesadores, un programa de un solo subproceso solo puede utilizar la mitad de los recursos de procesamiento disponibles.

Por otro lado, los programas con múltiples subprocesos activos pueden ejecutarse simultáneamente en múltiples procesadores. Cuando se diseñan correctamente, los programas de subprocesos múltiples pueden mejorar el rendimiento al utilizar los recursos de procesador disponibles de manera más efectiva.

El uso de múltiples subprocesos también puede ayudar a lograr un mejor rendimiento en sistemas de un solo procesador. Digamos, si un programa es de subproceso único, el procesador permanece inactivo mientras espera que se complete una operación de E/S síncrona. En un programa de subprocesos múltiples, aún se puede ejecutar otro subproceso mientras el primer subproceso está esperando que se complete la E/S, lo que permite que la aplicación siga progresando durante el bloqueo de E/S. En un escenario en tiempo real, esto es como leer el periódico mientras espera que hierva el agua, en lugar de esperar a que hierva el agua antes de comenzar a leer.

HILOS EN .NET



Cuando se inicia una nueva aplicación en Windows, crea un proceso para la aplicación con una identificación de proceso y algunos recursos se asignan a este nuevo proceso. Cada proceso contiene al menos un subproceso principal que se ocupa del punto de entrada de la ejecución de la aplicación. Un solo subproceso puede tener solo una ruta de ejecución, pero puede necesitar múltiples rutas de ejecución, y ahí es donde los subprocesos juegan un papel.

En .NET Core, Common Language Runtime (CLR) juega un papel importante en la creación y administración de los ciclos de vida de los subprocesos. En una nueva aplicación .NET Core, CLR crea un único subproceso en primer plano para ejecutar el código de la aplicación a través del método Main. Este subproceso se denomina subproceso principal. Junto con este hilo principal, un proceso puede crear uno o más hilos para ejecutar una parte del código.

Un programa también puede usar la clase ThreadPool para ejecutar código en subprocesos de trabajo que administra el CLR. Un programa C# es de un solo subproceso por diseño, esto significa que solo ejecuta una ruta del código a la vez. El punto de entrada de un programa C# comienza en el método Main, que es la ruta del subproceso principal.

Estados de un hilo

1. No iniciado: cuando se crea una instancia de una clase Thread, está en estado no iniciado, lo que significa que el subproceso aún no ha comenzado a ejcutarse. O, en otras palabras, no se ha llamado al método Start().

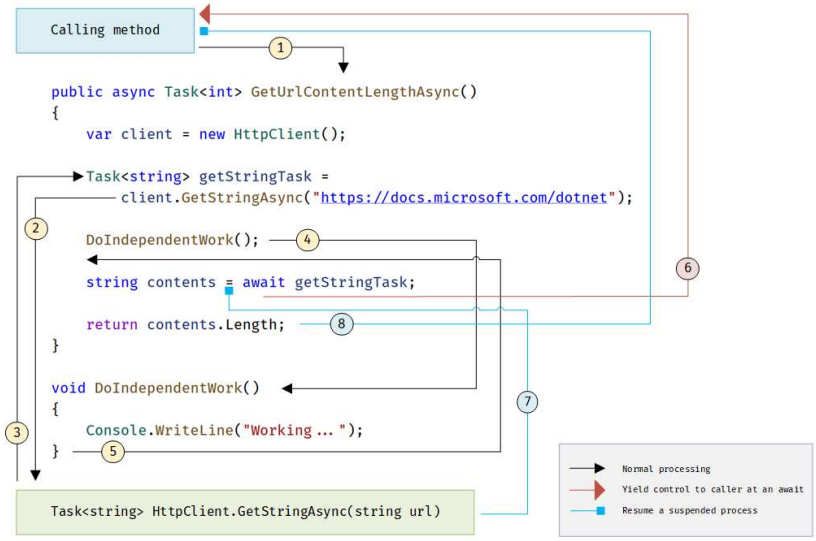
2. Ejecutable: un subproceso que está listo para ejecutarse se mueve al estado ejecutable. En este estado, un subproceso podría estar ejecutándose o podría estar listo para ejecutarse en cualquier momento. Es responsabilidad del *scheduler* de subprocesos darle al subproceso tiempo para ejecutarse. O en otras palabras, se llama al método Start().

3. En ejecución: un subproceso que se está ejecutando. O en otras palabras, el hilo obtiene el procesador.

4. No ejecutable: un subproceso puede estar en este estado porque se llamo a alguno de los siguientes metodos: *Sleep(), Wait() o Suspend()* o, porque está esperando una operación de E/S.

5. Muerto: se pueden utilizar las propiedades ThreadState o IsAlive de la clase Thread para conocer el estado de un hilo o subproceso.

HILOS VS ASYNC/AWAIT EN C#



Los métodos asíncronos están destinados a ser operaciones sin bloqueo. Una expresión de espera en un método asíncrono no bloquea el subproceso actual mientras se ejecuta la tarea esperada. En su lugar, la expresión registra el resto del método como una continuación y devuelve el control a la persona que llama al método asíncrono.

Las palabras clave async y await no provocan la creación de subprocesos adicionales. Los métodos asíncronos no requieren subprocesos múltiples porque un método asíncrono no se ejecuta en su propio subproceso. El método se ejecuta en el contexto de sincronización actual y usa el tiempo en el subproceso solo cuando el método está activo. Puede usar Task.Run para mover el trabajo vinculado a la CPU a un subproceso en segundo plano, pero un subproceso en segundo plano no ayuda con un proceso que solo espera que los resultados estén disponibles.

Ejemplo de como funciona un método asíncrono:

1. Un método llama y espera el método asincrónico GetUrlContentLengthAsync.

2. GetUrlContentLengthAsync crea una instancia de HttpClient y llama al método

asíncrono GetStringAsync para descargar el contenido de un sitio web como una cadena.
3. Algo sucede en GetStringAsync que suspende su progreso. Tal vez deba esperar a que se descargue un sitio web o alguna otra actividad de bloqueo. Para evitar el bloqueo de recursos, GetStringAsync cede el control a quien llama, GetUrlContentLengthAsync.

GetStringAsync devuelve Task<TResult>, donde TResult es una cadena, y GetUrlContentLengthAsync asigna la tarea a la variable getStringTask. La tarea representa el proceso en curso para la llamada a GetStringAsync, con el compromiso de producir un valor de cadena real cuando se complete el trabajo.

4. Debido a que getStringTask aún no se ha esperado, GetUrlContentLengthAsync puede continuar con otro trabajo que no depende del resultado final de GetStringAsync. Ese trabajo está representado por una llamada al método síncrono DoIndependentWork.

5. DoIndependentWork es un método síncrono que hace su trabajo y regresa a su llamador

6. GetUrlContentLengthAsync se ha quedado sin trabajo que puede hacer sin un resultado de getStringTask. GetUrlContentLengthAsync luego quiere calcular y devolver la longitud de la cadena descargada, pero el método no puede calcular ese valor hasta que el método tenga la cadena.

Por lo tanto, GetUrlContentLengthAsync usa un operador de espera para suspender su progreso y ceder el control al método que llamó a GetUrlContentLengthAsync. GetUrlContentLengthAsync devuelve Task<int> a la persona que llama. La tarea representa una promesa de producir un resultado entero que es la longitud de la cadena descargada.

Note
Si GetStringAsync (y, por lo tanto, getStringTask) se completa antes de que GetUrlContentLengthAsync lo espere, el control permanece en GetUrlContentLengthAsync. El gasto de suspender y luego volver a GetUrlContentLengthAsync se perdería si el proceso asíncrono llamado getStringTask ya se completó y GetUrlContentLengthAsync no tiene que esperar el resultado final.

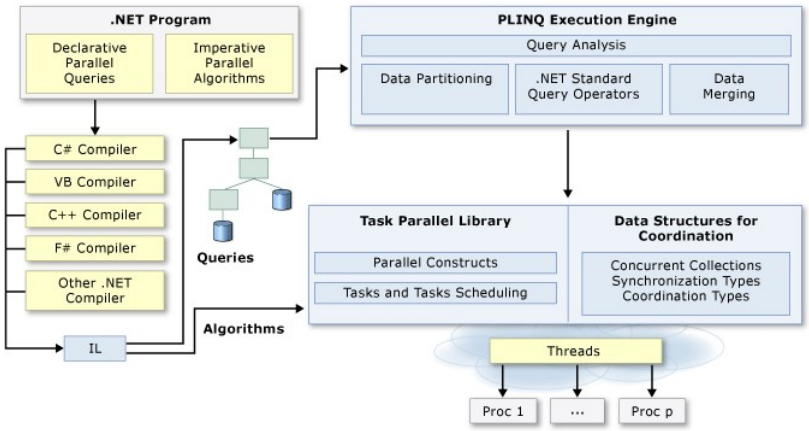
Dentro del método de llamada, el patrón de procesamiento continúa. La persona que llama puede hacer otro trabajo que no depende del resultado de GetUrlContentLengthAsync antes de esperar ese resultado, o la persona que llama puede esperar inmediatamente. El método de llamada está esperando

GetUrlContentLengthAsync y GetUrlContentLengthAsync está esperando GetStringAsync.

7. GetStringAsync completa y produce un resultado de cadena. La llamada a GetStringAsync no devuelve el resultado de la cadena de la forma esperada. (Recuerde que el método ya devolvió una tarea en el paso 3). En su lugar, el resultado de la cadena se almacena en la tarea que representa la finalización del método, getStringTask. El operador await recupera el resultado de getStringTask. La declaración de asignación asigna el resultado recuperado a los contenidos.

8. Cuando GetUrlContentLengthAsync tiene el resultado de la cadena, el método puede calcular la longitud de la cadena. Luego, el trabajo de GetUrlContentLengthAsync también está completo y el controlador de eventos en espera puede reanudarse. En el ejemplo completo al final del tema, puede confirmar que el controlador de eventos recupera e imprime el valor del resultado de longitud. Si es nuevo en la programación asincrónica, tómese un minuto para considerar la diferencia entre el comportamiento sincrónico y asincrónico. Un método síncrono regresa cuando su trabajo está completo (paso 5), pero un método asíncrono regresa un valor de tarea cuando su trabajo está suspendido (pasos 3 y 6). Cuando el método asincrónico finalmente completa su trabajo, la tarea se marca como completada y el resultado, si lo hay, se almacena en la tarea.

TASK PARALLEL LIBRARY (TPL)



TASK PARALLEL LIBRARY (TPL) es un conjunto de tipos públicos y API en los namespaces System.Threading y System.Threading.Tasks. El propósito de TPL es hacer que los desarrolladores sean más productivos al simplificar el proceso de agregar paralelismo y concurrencia a las aplicaciones. El TPL escala dinámicamente el grado de concurrencia para usar todos los procesadores disponibles de la manera más eficiente. Además, TPL maneja la partición del trabajo, la programación de subprocesos en ThreadPool, el soporte de cancelación, la gestión de estado y otros detalles de bajo nivel.

TPL se basa en el concepto de una tarea, que representa una operación asíncrona. En cierto modo, una tarea se asemeja a un hilo o elemento de trabajo ThreadPool pero en un nivel más alto de abstracción. El término paralelismo de tareas se refiere a una o más tareas independientes que se ejecutan simultáneamente. Las tareas proporcionan dos beneficios principales:

1. Uso más eficiente y escalable de los recursos del sistema.

Detrás de escena, las tareas se ponen en cola en ThreadPool, que se ha mejorado con algoritmos que determinan y se ajustan a la cantidad de subprocesos. Estos

algoritmos proporcionan equilibrio de carga para maximizar el rendimiento. Este proceso hace que las tareas sean relativamente livianas y puede crear muchas de ellas para permitir un paralelismo detallado.

2. Más control programático del que es posible con un subproceso.

Las tareas y el marco creado a su alrededor brindan un amplio conjunto de API que admiten espera, cancelación, continuaciones, manejo sólido de excepciones, estado detallado, programación personalizada y más.

Referencia: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl

PROGRAMACIÓN CONCURRENTE

```
public class Counting { public static void
main(String[] args) throws
InterruptedException { class Counter { int
counter = 0; public void increment() {
counter++; } public int get() { return
counter; } } final Counter counter = new
Counter(); class CountingThread extends
Thread { public void run() { for (int x =
0; x < 500000; x++) { counter.increment();
} } } CountingThread t1 = new
CountingThread(); CountingThread t2 = new
CountingThread(); t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter.get()); } }
```

```
java Counting 553706
java Counting 547818
java Counting 613014
```

Veamos un ejemplo simple con un contador y dos hilos que lo aumentan. El programa no debería ser demasiado complicado. Tenemos un objeto que contiene un contador que aumenta con el método increment() y lo recupera con el método get() y dos hilos que lo aumentan.

El programa aumenta el contador en un lugar, en el método de aumento que utiliza el comando contador++. Si analizamos el detalle de lo que ocurre de fondo, veríamos que la ejecución consta de varias partes:

1. Leer el valor del contador de la memoria
2. Aumentar el valor localmente
3. Almacenar el valor del contador en la memoria

Ahora podemos imaginar lo que puede salir mal en esta secuencia. Si tenemos dos subprocesos que aumentan el contador de forma independiente, entonces podríamos tener este escenario:

1. El valor del contador es 115
2. El primer hilo lee el valor del contador de la memoria (115)
3. El primer subproceso aumenta el valor del contador local (116)
4. El segundo hilo lee el valor del contador de la memoria (115)

5. El segundo hilo aumenta el valor del contador local (116)
6. El segundo hilo guarda el valor del contador local en la memoria (116)
7. El primer hilo guarda el valor del contador local en la memoria (116)
8. El valor del contador es 116