**Exploración del Modelo de Programación Asincrónica Basado en Tareas (TAP) en C# con async y await**

Ing. Anibal Guillermo Martínez Arcia

Junio 29, 2025

**Resumen**

Desarrollar Software en a actualidad, exige aplicaciones altamente reactivas y escalables, lo que destaca la relevancia de la programación asincrónica. Este artículo presenta una exploración detallada del Modelo de Programación Asincrónica Basado en Tareas (TAP) en C#, enfocándose en el empleo de las palabras clave async y await para la gestión eficiente de operaciones que consumen tiempo sin bloquear el hilo principal de ejecución. A través de una analogía práctica de la preparación de un desayuno y la implementación de ejemplos de código en C#, se comparan las aproximaciones sincrónica y asincrónica. Los resultados demuestran cómo la concurrencia de tareas, posibilitada por el inicio simultáneo de operaciones independientes, reduce significativamente el tiempo total de procesamiento. Se concluye que la correcta implementación del modelo TAP es crucial para optimizar la eficiencia, la capacidad de respuesta y la experiencia de usuario en aplicaciones modernas.

**Palabras clave.**

Async / Away, Modelo TAP, Concurrencia, Rendimiento, Programación asíncrona.

**Introducción**

Uno de los mayores desafíos que enfrentamos al desarrollar Software hoy en día, es el de construir aplicaciones reactivas, eficientes y escalables. En un entorno donde las operaciones de entrada/salida (E/S), las llamadas a servicios de red o el acceso a bases de datos pueden introducir latencias significativas, la programación sincrónica tradicional se convierte en un cuello de botella. Bajo este modelo, el hilo principal de ejecución se bloquea mientras espera la finalización de cada operación, lo que resulta en interfaces de usuario congeladas en aplicaciones cliente y en la saturación de hilos en aplicaciones de servidor, comprometiendo gravemente la capacidad de respuesta y la eficiencia de los recursos.

Para abordar estas limitaciones, ha surgido y evolucionado la programación asincrónica, permitiendo que las aplicaciones continúen procesando otras tareas mientras ciertas operaciones de larga duración se ejecutan en segundo plano. Dentro del ecosistema .NET, el Modelo de Programación Asincrónica Basado en Tareas (TAP), implementado a través de las palabras clave async y await, representa una abstracción poderosa y altamente legible para manejar la concurrencia. Este modelo transforma la complejidad subyacente de la gestión de hilos y callbacks en un flujo de código que se asemeja a la programación secuencial, facilitando su comprensión y mantenimiento.

El presente artículo tiene como objetivo principal explorar en profundidad el modelo TAP, detallando cómo async y await permiten construir programas asincrónicos de forma intuitiva. Se examinará la diferencia fundamental entre asincronía y paralelismo y se demostrará la optimización del rendimiento mediante la ejecución concurrente de tareas independientes. Para ello, se utilizará una analogía práctica y se proporcionarán ejemplos de código en C# que ilustran la evolución desde un enfoque sincrónico hasta una implementación asincrónica optimizada.

El documento se estructura de la siguiente manera: primero, se presentarán los principios básicos del modelo TAP; seguidamente, se establecerá una comparación entre la ejecución sincrónica y asincrónica mediante un escenario de ejemplo. Posteriormente, se abordará la optimización de la ejecución de tareas a través de la concurrencia. Finalmente, se discutirán las implicaciones de la composición de tareas asincrónicas y se presentarán las conclusiones derivadas de este análisis.

**Metodología**

Para ilustrar de manera práctica y comparativa los beneficios del Modelo de Programación Asincrónica Basado en Tareas (TAP) en C#, se diseñó un estudio de caso analógico centrado en una secuencia de tareas cotidianas. Este enfoque permite abstraer la complejidad inherente a las operaciones de software de larga duración (como E/S de red o disco) a un escenario intuitivo y fácilmente comprensible.

### Escenario de Aplicación. La Preparación del Desayuno

El caso de estudio se basa en la analogía de la preparación de un desayuno, que implica una serie de tareas con diferentes duraciones y dependencias:

* Servir café.
* Freír huevos.
* Freír tocino.
* Tostar pan.
* Untar mantequilla y mermelada en la tostada.
* Servir jugo de naranja.

Esta analogía es particularmente útil porque refleja tanto operaciones que pueden ejecutarse concurrentemente (como freír huevos y tostar pan) como operaciones secuenciales que dependen de la finalización de una tarea previa (untar mantequilla solo después de que el pan esté tostado).

### Implementación del Caso de Estudio

La analogía del desayuno fue implementada en C# utilizando clases representativas para cada componente (Coffee, Egg, Bacon, Toast, Juice). Las operaciones que implican "tiempos de espera" (freír, tostar) fueron simuladas mediante la función Task.Delay(), que introduce pausas no bloqueantes, permitiendo así la simulación de la asincronía. Las operaciones rápidas (servir café, servir jugo, untar mantequilla/mermelada) se implementaron como métodos sincrónicos.

Se desarrollaron y analizaron tres versiones principales del código para comparar su eficiencia y el impacto en el flujo de ejecución:

1. Versión Sincrónica: Representa el enfoque tradicional donde cada tarea se ejecuta de principio a fin antes de que la siguiente pueda comenzar. Todas las operaciones simuladas con Task.Delay() se bloquearon explícitamente usando .Wait() para emular el comportamiento sincrónico.
2. Versión Asincrónica Básica (Secuencial): Introduce las palabras clave async y await en los métodos de tareas de larga duración. Sin embargo, las llamadas await se realizan inmediatamente después de iniciar cada tarea, lo que, aunque no bloquea el hilo, aún mantiene una secuencia lógica de ejecución similar a la sincrónica en términos de tiempo total. Esta versión muestra el beneficio de no bloquear el hilo, pero sin explotar la concurrencia.
3. Versión Asincrónica Optimizada (Concurrente): Esta versión maximiza la asincronía iniciando múltiples tareas de larga duración (FryEggsAsync, FryBaconAsync, ToastBreadAsync) de forma simultánea al comienzo del programa. Las expresiones await se posponen hasta el momento en que sus resultados son realmente necesarios, permitiendo que las operaciones se superpongan en el tiempo y reduciendo el tiempo total de ejecución. Adicionalmente, se demostró la composición de tareas asincrónicas mediante la creación de un método (MakeToastWithButterAndJamAsync) que encapsula una operación asincrónica seguida de operaciones sincrónicas.

El código fuente completo para cada versión fue desarrollado y ejecutado en el entorno de desarrollo .NET para C#, utilizando una aplicación de consola. La medición de los "tiempos de cocción" se basa en las pausas simuladas por Task.Delay(), que sirven como métrica relativa para comparar la eficiencia de cada enfoque.

El modelo de Programación Asincrónica basado en Tareas (Task-based Asynchronous Programming, TAP) proporciona una capa de abstracción sobre la programación asincrónica tradicional. Este modelo permite escribir código como una secuencia de instrucciones, tal como se hace en la programación sincrónica. La diferencia es que, aunque el código parece ejecutarse línea por línea, en realidad algunas instrucciones inician operaciones que continúan ejecutándose en segundo plano, sin bloquear el flujo principal.

Para lograr esto, el compilador transforma los métodos marcados con la palabra clave async en una máquina de estados que administra las tareas (Task) de manera eficiente. Algunas instrucciones dentro del método pueden iniciar trabajo asincrónico y devolver un objeto Task que representa esa operación en curso. El compilador se encarga de pausar y reanudar la ejecución en los puntos necesarios, sin que el programador tenga que escribir código complejo para manejar hilos manualmente.

El objetivo principal de TAP es permitir que el código asincrónico se lea de forma clara y secuencial, aunque internamente su ejecución dependa de factores externos, como el tiempo de respuesta de una operación de E/S, una llamada a red, o el acceso a una base de datos.

Una buena forma de entender este modelo es mediante una analogía con tareas cotidianas. Considera, por ejemplo, las instrucciones para preparar un desayuno. Algunas tareas se pueden hacer al mismo tiempo, y otras deben esperar a que una tarea previa se complete. Usando async y await, se puede escribir código que refleje esa lógica de manera simple y legible. Las instrucciones podrían verse así:

1. Servir una taza de café.
2. Calentar una sartén y luego freír dos huevos.
3. Freír tres rebanadas de tocino.
4. Tostar dos rebanadas de pan.
5. Untar mantequilla y mermelada en la tostada.
6. Servir un vaso de jugo de naranja.

Este ejemplo muestra cómo varias tareas pueden realizarse en paralelo (como tostar pan y freír tocino), mientras otras dependen del resultado anterior (untar mantequilla solo después de que el pan esté tostado). De manera similar, en C# el uso de async y await permite expresar estas dependencias de forma clara, sin necesidad de complicarse con hilos, eventos o callbacks.

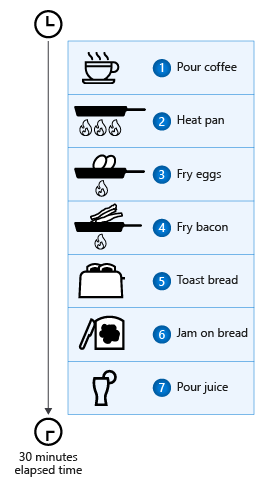
Si tienes experiencia en la cocina, probablemente puedas ejecutar varias de las instrucciones del desayuno de forma asincrónica. Por ejemplo, puedes comenzar a calentar la sartén para los huevos y, mientras se calienta, empezar a freír el tocino. Luego, puedes poner el pan en la tostadora y volver a los huevos para comenzar a cocinarlos. En cada paso, inicias una tarea y luego cambias a otra que ya está lista para continuar.

Este ejemplo de cocinar el desayuno ilustra claramente cómo funciona la asincronía. No es necesario que varias personas trabajen al mismo tiempo (no es paralelismo), sino que una sola persona o un único hilo de ejecución puede avanzar en múltiples tareas al cambiar de una a otra cuando están listas para recibir atención. Cada actividad (freír, tostar, calentar) progresa por su cuenta una vez que ha sido iniciada, sin necesidad de que la supervises constantemente.

La asincronía, entonces, permite que una sola unidad de trabajo maneje múltiples tareas de forma eficiente, sin esperar a que cada una finalice antes de empezar la siguiente. Esto es útil cuando hay tiempos de espera, como el calentamiento de una sartén o el tostado del pan.

En contraste, el paralelismo implica que varias personas o varios hilos ejecutan tareas simultáneamente. Por ejemplo, una persona cocina los huevos, otra fríe el tocino, y una tercera prepara la tostada. Cada una está dedicada a una tarea específica y trabaja en paralelo con las demás. En este caso, cada persona (o hilo) permanece atenta exclusivamente a su tarea, bloqueada hasta que se complete: esperando dar la vuelta al tocino, sacar el pan cuando salte la tostadora, etc.

**Figura 1.**

*****Tareas ejecutándose sincrónicamente*

Nota: Recuperado de: <https://learn.microsoft.com/es-es/dotnet/csharp/asynchronous-programming/>

Pensemos cómo se vería esa lista si la escribiéramos en código C# que ejecuta cada tarea una tras otra, sin usar asincronía:

using System;

using System.Threading.Tasks;

namespace AsyncBreakfast

{

// Estas clases están intencionalmente vacías para fines de este ejemplo.

// Son simplemente clases marcadoras para demostración, no contienen propiedades

// y no tienen otro propósito.

internal class Bacon { }

internal class Coffee { }

internal class Egg { }

internal class Juice { }

internal class Toast { }

class Program

{

static void Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("el café está listo");

Egg eggs = FryEggs(2);

Console.WriteLine("los huevos están listos");

Bacon bacon = FryBacon(3);

Console.WriteLine("el tocino está listo");

Toast toast = ToastBread(2);

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("las tostadas están listas");

Juice oj = PourOJ();

Console.WriteLine("el jugo de naranja está listo");

Console.WriteLine("¡El desayuno está listo!");

}

private static Juice PourOJ()

{

Console.WriteLine("Sirviendo jugo de naranja");

return new Juice();

}

private static void ApplyJam(Toast toast) =>

Console.WriteLine("Poniendo mermelada en la tostada");

private static void ApplyButter(Toast toast) =>

Console.WriteLine("Poniendo mantequilla en la tostada");

private static Toast ToastBread(int slices)

{

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("Poniendo una rebanada de pan en la tostadora");

}

Console.WriteLine("Comenzando a tostar...");

Task.Delay(3000).Wait();

Console.WriteLine("Sacando las tostadas de la tostadora");

return new Toast();

}

private static Bacon FryBacon(int slices)

{

Console.WriteLine($"poniendo {slices} rebanadas de tocino en la sartén");

Console.WriteLine("cocinando el primer lado del tocino...");

Task.Delay(3000).Wait();

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("volteando una rebanada de tocino");

}

Console.WriteLine("cocinando el segundo lado del tocino...");

Task.Delay(3000).Wait();

Console.WriteLine("Poner el tocino en el plato");

return new Bacon();

}

private static Egg FryEggs(int howMany)

{

Console.WriteLine("Calentando la sartén para huevos...");

Task.Delay(3000).Wait();

Console.WriteLine($"rompiendo {howMany} huevos");

Console.WriteLine("cocinando los huevos...");

Task.Delay(3000).Wait();

Console.WriteLine("Poner los huevos en el plato");

return new Egg();

}

private static Coffee PourCoffee()

{

Console.WriteLine("Sirviendo café");

return new Coffee();

}

}

}

Supongamos que esas instrucciones del desayuno se ejecutan de forma estrictamente sincrónica, como lo haría un ordenador interpretando cada línea de código secuencialmente. En ese caso, la preparación completa tardaría alrededor de 30 minutos, ya que el tiempo total sería la suma exacta del tiempo que toma cada tarea individual.

Esto se debe a que, en un modelo sincrónico, la ejecución de una instrucción no comienza hasta que la anterior ha finalizado completamente. El flujo se bloquea en cada paso, esperando que la tarea actual concluya antes de continuar con la siguiente. Como resultado, se desperdicia tiempo valioso que podría aprovecharse ejecutando otras tareas en paralelo o en espera activa. En el contexto del desayuno, este enfoque sincrónico generaría una experiencia insatisfactoria: los huevos podrían estar fríos mientras se espera a que el pan esté listo, y el jugo no se serviría sino hasta el final.

Para que la computadora pueda ejecutar estas tareas de manera más eficiente, es necesario escribir código asincrónico. En el desarrollo de aplicaciones cliente, esto es especialmente importante para mantener una interfaz de usuario reactiva: la aplicación no debe congelarse mientras se descarga información desde internet. En aplicaciones de servidor, los hilos deben permanecer disponibles para atender múltiples solicitudes, en lugar de quedar bloqueados esperando la finalización de operaciones de entrada/salida.

El uso de código sincrónico en contextos donde existen alternativas asincrónicas puede afectar gravemente el rendimiento y la escalabilidad. En escenarios con alta demanda, cada hilo bloqueado representa un costo de recursos y una pérdida de capacidad para procesar otras tareas de forma eficiente.

Hoy en día, las aplicaciones modernas bien diseñadas requieren programación asincrónica. Sin soporte nativo del lenguaje, esto implicaría utilizar estructuras complejas como callbacks, eventos o manejadores de finalización, lo que tiende a dificultar la lectura y comprensión del código. La ventaja del código sincrónico tradicional es que resulta fácil de leer gracias a su flujo paso a paso. Sin embargo, los modelos asincrónicos antiguos exigían que el desarrollador se concentrara más en la gestión del flujo asincrónico que en la lógica del programa.

### No bloquees; espera de forma inteligente

Una mala práctica común es escribir código sincrónico para operaciones que deberían realizarse de manera asincrónica. Este enfoque bloquea el hilo actual, impidiendo que se lleven a cabo otras tareas mientras una operación está en curso. Esto es similar a quedarse mirando la tostadora después de poner el pan: no haces nada más mientras esperas a que el pan salte. No aprovechas ese tiempo para sacar la mantequilla y la mermelada, o para vigilar la sartén en la estufa.

Del mismo modo, en tu código, deberías permitir que otras tareas avancen mientras una operación está pendiente. Para eso existe await, una palabra clave que permite iniciar una tarea de forma asincrónica y suspender temporalmente la ejecución del método sin bloquear el hilo. Cuando la tarea se completa, la ejecución continúa automáticamente desde el punto en el que fue suspendida.

A continuación, veremos cómo se vería una versión asincrónica básica del código que representa la preparación del desayuno:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

Egg eggs = await FryEggsAsync(2);

Console.WriteLine("eggs are ready");

Bacon bacon = await FryBaconAsync(3);

Console.WriteLine("bacon is ready");

Toast toast = await ToastBreadAsync(2);

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

}

En esta etapa, el código ha sido modificado para que los métodos FryEggs, FryBacon y ToastBread ahora devuelvan objetos Task<Egg>, Task<Bacon> y Task<Toast>, respectivamente. Además, los nombres de estos métodos se han actualizado siguiendo la convención estándar en C#, agregando el sufijo Async: FryEggsAsync, FryBaconAsync y ToastBreadAsync.

El método Main también ha cambiado: ahora devuelve un objeto Task, aunque no contiene una expresión return. Esto es perfectamente válido y forma parte del diseño del modelo asincrónico de C#. (Para más detalles, puedes consultar el tema *Evaluación de una función asincrónica que devuelve void* en la documentación oficial.)

**Nota**:  
Aunque el código se ha adaptado para usar async y await, todavía no se aprovechan por completo las ventajas de la programación asincrónica. El tiempo de ejecución sigue siendo similar al de la versión sincrónica original, ya que las tareas aún se procesan una tras otra, sin solaparse. Más adelante, veremos una versión optimizada que demuestra mejor las capacidades del enfoque asincrónico.

Volviendo a la analogía del desayuno, imagina que has comenzado a usar herramientas asincrónicas, pero tu comportamiento sigue siendo sincrónico. Por ejemplo, comienzas a freír los huevos y, aunque el hilo no está bloqueado, no inicias ninguna otra tarea hasta que los huevos estén listos. Luego haces lo mismo con el tocino y la tostada. En otras palabras, aún estás mirando la tostadora sin hacer otra cosa, pero ahora al menos podrías responder a interrupciones si surgieran.

En este nuevo modelo, el subproceso que ejecuta la lógica del desayuno no queda bloqueado mientras las tareas están en proceso. Para algunas aplicaciones, como aquellas con interfaces gráficas, este cambio ya es suficiente: la aplicación sigue siendo receptiva mientras se descargan datos o se realizan llamadas a servicios web. Pero en otros escenarios, es más eficiente iniciar otras tareas independientes mientras esperas que finalicen las actuales.

### Iniciar tareas de forma simultánea

En muchas aplicaciones, es recomendable lanzar varias tareas independientes de inmediato, especialmente si no dependen entre sí. Esto permite que el sistema utilice mejor el tiempo disponible, iniciando nuevas tareas mientras otras siguen en ejecución. Al aplicar esta lógica al ejemplo del desayuno, puedes cocinar los huevos, freír el tocino y tostar el pan al mismo tiempo. Así, todos los elementos del desayuno estarán listos aproximadamente a la misma hora, lo que se traduce en una mejor experiencia (y un desayuno caliente).

La clase System.Threading.Tasks.Task y sus tipos relacionados permiten aplicar este modelo en el código. Este enfoque refleja más fielmente cómo se manejan las tareas en el mundo real: comienzas varias operaciones al mismo tiempo, y luego vas atendiendo cada una cuando necesita tu intervención.

En términos de código, esto implica iniciar cada tarea por separado y conservar la referencia al objeto Task que representa cada operación. Más adelante, puedes usar await sobre esas tareas para continuar la ejecución cuando sus resultados estén disponibles.

### Implementación: primer paso

Para comenzar con esta mejora, debes almacenar las tareas que se inician, en lugar de usar await inmediatamente. Por ejemplo:

Task<Egg> eggsTask = FryEggsAsync(2);

Task<Bacon> baconTask = FryBaconAsync(3);

Task<Toast> toastTask = ToastBreadAsync(2);

Una vez iniciadas, puedes esperar sus resultados cuando sea necesario:

Egg eggs = await eggsTask;

Bacon bacon = await baconTask;

Toast toast = await toastTask;

De este modo, todas las tareas se ejecutan simultáneamente, y el hilo solo espera cuando realmente necesita los resultados:

using System;

using System.Threading.Tasks;

namespace AsyncBreakfast

{

// Estas clases están intencionalmente vacías para fines de este ejemplo.

// Son simplemente clases marcadoras para demostración, no contienen propiedades

// y no tienen otro propósito.

internal class Bacon { }

internal class Coffee { }

internal class Egg { }

internal class Juice { }

internal class Toast { }

class Program

{

// Marcamos Main como async Task para poder usar await dentro de él.

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("el café está listo");

// --- INICIO DE LA MODIFICACIÓN ---

// Iniciar la tarea de freír huevos. La tarea comienza a ejecutarse

// inmediatamente, pero no esperamos su finalización todavía.

Task<Egg> eggsTask = FryEggsAsync(2);

// Iniciar la tarea de freír tocino. También comienza a ejecutarse

// en paralelo con la tarea de los huevos.

Task<Bacon> baconTask = FryBaconAsync(3);

// Iniciar la tarea de tostar pan. También se ejecuta en paralelo.

Task<Toast> toastTask = ToastBreadAsync(2);

// --- FIN DE LA MODIFICACIÓN (primera parte) ---

// Ahora 'await'amos las tareas que toman tiempo, una por una,

// pero las tareas ya estaban en progreso desde antes.

Egg eggs = await eggsTask;

Console.WriteLine("los huevos están listos");

Bacon bacon = await baconTask;

Console.WriteLine("el tocino está listo");

Toast toast = await toastTask;

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("las tostadas están listas");

Juice oj = PourOJ();

Console.WriteLine("el jugo de naranja está listo");

Console.WriteLine("¡El desayuno está listo!");

}

private static Juice PourOJ()

{

Console.WriteLine("Sirviendo jugo de naranja");

return new Juice();

}

private static void ApplyJam(Toast toast) =>

Console.WriteLine("Poniendo mermelada en la tostada");

private static void ApplyButter(Toast toast) =>

Console.WriteLine("Poniendo mantequilla en la tostada");

// Convertimos este método a async Task<Toast> y usamos Task.Delay sin .Wait()

private static async Task<Toast> ToastBreadAsync(int slices)

{

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("Poniendo una rebanada de pan en la tostadora");

}

Console.WriteLine("Comenzando a tostar...");

await Task.Delay(3000); // Ahora esperamos de forma no bloqueante

Console.WriteLine("Sacando las tostadas de la tostadora");

return new Toast();

}

// Convertimos este método a async Task<Bacon> y usamos Task.Delay sin .Wait()

private static async Task<Bacon> FryBaconAsync(int slices)

{

Console.WriteLine($"poniendo {slices} rebanadas de tocino en la sartén");

Console.WriteLine("cocinando el primer lado del tocino...");

await Task.Delay(3000); // Ahora esperamos de forma no bloqueante

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("volteando una rebanada de tocino");

}

Console.WriteLine("cocinando el segundo lado del tocino...");

await Task.Delay(3000); // Ahora esperamos de forma no bloqueante

Console.WriteLine("Poner el tocino en el plato");

return new Bacon();

}

// Convertimos este método a async Task<Egg> y usamos Task.Delay sin .Wait()

private static async Task<Egg> FryEggsAsync(int howMany)

{

Console.WriteLine("Calentando la sartén para huevos...");

await Task.Delay(3000); // Ahora esperamos de forma no bloqueante

Console.WriteLine($"rompiendo {howMany} huevos");

Console.WriteLine("cocinando los huevos...");

await Task.Delay(3000); // Ahora esperamos de forma no bloqueante

Console.WriteLine("Poner los huevos en el plato");

return new Egg();

}

private static Coffee PourCoffee()

{

Console.WriteLine("Sirviendo café");

return new Coffee();

}

}

## }

Con los cambios anteriores, el código ya inicia las tareas asincrónicas para preparar los huevos, el tocino y las tostadas. Sin embargo, aún no se logra una mejora real en el tiempo total de preparación del desayuno. ¿Por qué?

Porque las expresiones await se aplican inmediatamente después de iniciar cada tarea. Esto significa que, aunque el método sea asincrónico, las tareas siguen ejecutándose de forma secuencial desde el punto de vista lógico: el programa espera a que se terminen los huevos antes de iniciar la siguiente espera, y así sucesivamente.

### Optimización: posponer los await

Para optimizar realmente el rendimiento, el siguiente paso consiste en iniciar todas las tareas lo antes posible sin esperarlas de inmediato y luego aplicar las expresiones await al final del método, justo antes de necesitar sus resultados.

Este cambio permite que las tareas se ejecuten en paralelo mientras el método principal sigue su curso. Volviendo al ejemplo del desayuno: ahora comienzas a freír los huevos, luego el tocino, luego la tostada, y solo al final te detienes a recoger el resultado de cada preparación para servir el desayuno completo.

En términos de código, el patrón sería el siguiente:

Task<Egg> eggsTask = FryEggsAsync(2);

Task<Bacon> baconTask = FryBaconAsync(3);

Task<Toast> toastTask = ToastBreadAsync(2);

// Otras operaciones que no dependen directamente de los resultados anteriores

PourCoffee();

PourOrangeJuice();

// Al final, esperamos los resultados cuando los necesitamos

Egg eggs = await eggsTask;

Bacon bacon = await baconTask;

Toast toast = await toastTask;

Este enfoque aprovecha al máximo la asincronía: las tareas comienzan lo más pronto posible y el hilo no se bloquea innecesariamente. Así, los tiempos de espera se superponen, lo que reduce significativamente el tiempo total de ejecución:

using System;

using System.Threading.Tasks;

namespace AsyncBreakfast

{

// Estas clases están intencionalmente vacías para fines de este ejemplo.

// Son simplemente clases marcadoras para demostración, no contienen propiedades

// y no tienen otro propósito.

internal class Bacon { }

internal class Coffee { }

internal class Egg { }

internal class Juice { }

internal class Toast { }

class Program

{

// Marcamos Main como async Task para poder usar await dentro de él.

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("el café está listo");

// Iniciar todas las tareas de cocción en paralelo

Task<Egg> eggsTask = FryEggsAsync(2);

Task<Bacon> baconTask = FryBaconAsync(3);

Task<Toast> toastTask = ToastBreadAsync(2);

// Preparar las tostadas (esperando solo por las tostadas)

Toast toast = await toastTask;

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("las tostadas están listas");

// Servir el jugo de naranja (puede hacerse mientras lo demás cocina)

Juice oj = PourOJ();

Console.WriteLine("el jugo de naranja está listo");

// Ahora, y solo ahora, esperamos que los huevos y el tocino estén listos.

// Es muy probable que ya hayan terminado o estén a punto de terminar,

// ya que se iniciaron al principio junto con las tostadas.

Egg eggs = await eggsTask;

Console.WriteLine("los huevos están listos");

Bacon bacon = await baconTask;

Console.WriteLine("el tocino está listo");

Console.WriteLine("¡El desayuno está listo!");

}

private static Juice PourOJ()

{

Console.WriteLine("Sirviendo jugo de naranja");

return new Juice();

}

private static void ApplyJam(Toast toast) =>

Console.WriteLine("Poniendo mermelada en la tostada");

private static void ApplyButter(Toast toast) =>

Console.WriteLine("Poniendo mantequilla en la tostada");

private static async Task<Toast> ToastBreadAsync(int slices)

{

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("Poniendo una rebanada de pan en la tostadora");

}

Console.WriteLine("Comenzando a tostar...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine("Sacando las tostadas de la tostadora");

return new Toast();

}

private static async Task<Bacon> FryBaconAsync(int slices)

{

Console.WriteLine($"poniendo {slices} rebanadas de tocino en la sartén");

Console.WriteLine("cocinando el primer lado del tocino...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("volteando una rebanada de tocino");

}

Console.WriteLine("cocinando el segundo lado del tocino...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine("Poner el tocino en el plato");

return new Bacon();

}

private static async Task<Egg> FryEggsAsync(int howMany)

{

Console.WriteLine("Calentando la sartén para huevos...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine($"rompiendo {howMany} huevos");

Console.WriteLine("cocinando los huevos...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine("Poner los huevos en el plato");

return new Egg();

}

private static Coffee PourCoffee()

{

Console.WriteLine("Sirviendo café");

return new Coffee();

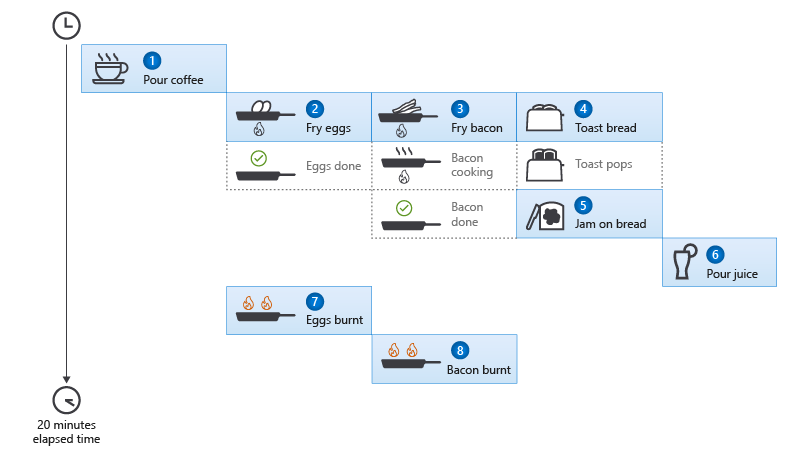
}

}

}

Ahora tiene un desayuno preparado de forma asincrónica que tarda unos 20 minutos en prepararse. El tiempo total de cocción se reduce porque algunas tareas se ejecutan simultáneamente.

**Figura 2.**

*Ejecución asíncrona*

Nota: Recuperado de: <https://learn.microsoft.com/es-es/dotnet/csharp/asynchronous-programming/>

El código actualiza el proceso de preparación del desayuno, reduciendo el tiempo total de cocción. Sin embargo, también introduce una posible regresión: los huevos y el tocino podrían terminar quemándose si no se gestionan adecuadamente los tiempos. El programa inicia todas las tareas asincrónicas al mismo tiempo y solo espera por cada una cuando necesita su resultado.

Este enfoque se asemeja al comportamiento de una aplicación web que realiza varias solicitudes a distintos microservicios. Primero lanza todas las solicitudes de forma paralela, y luego espera (await) los resultados para componer una única respuesta, como una página web completa.

### Composición de tareas

En las versiones anteriores del código, las tareas se inician de manera simultánea para preparar todo el desayuno, con excepción de la tostada. La preparación de la tostada es un ejemplo de **composición**: se trata de una combinación entre una operación asincrónica (tostar el pan) y operaciones sincrónicas (untar mantequilla y mermelada).

Este patrón permite ilustrar un principio importante de la programación asincrónica:

**Importante:**  
Cuando una operación asincrónica se combina con trabajo sincrónico, el resultado debe tratarse como una operación asincrónica. Dicho de otro modo, si alguna parte de un proceso es asincrónica, todo el proceso debe considerarse asincrónico.

Hasta este punto, hemos aprendido a usar objetos Task o Task<TResult> para almacenar tareas en ejecución y a esperar su finalización antes de usar los resultados. El siguiente paso consiste en crear métodos que representen tareas compuestas. Por ejemplo, antes de servir el desayuno, es necesario esperar a que finalice la operación de tostar el pan, y solo entonces aplicar la mantequilla y la mermelada.

Esto puede representarse con el siguiente código:

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number);

ApplyButter(toast);

ApplyJam(toast);

return toast;

}

El método MakeToastWithButterAndJamAsync incluye el modificador async en su declaración, lo cual le indica al compilador que el método contiene al menos una expresión await y que ejecuta operaciones asincrónicas. Este método encapsula el proceso completo de preparar la tostada: primero, tostar el pan de manera asincrónica; luego, aplicar mantequilla y mermelada de forma sincrónica.

El método devuelve un objeto Task<Toast>, que representa la tarea compuesta de ejecutar estas tres operaciones como una única unidad lógica de trabajo. Esta composición permite mantener la asincronía en toda la operación, respetando el principio de que **si una parte de una operación es asincrónica, toda la operación debe tratarse como asincrónica.**

La versión revisada del método Main ahora se ve así:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

var eggsTask = FryEggsAsync(2);

var baconTask = FryBaconAsync(3);

var toastTask = MakeToastWithButterAndJamAsync(2);

var eggs = await eggsTask;

Console.WriteLine("eggs are ready");

var bacon = await baconTask;

Console.WriteLine("bacon is ready");

var toast = await toastTask;

Console.WriteLine("toast is ready");

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

}

Código completo:

using System;

using System.Threading.Tasks;

namespace AsyncBreakfast

{

// Estas clases están intencionalmente vacías para fines de este ejemplo.

// Son simplemente clases marcadoras para demostración, no contienen propiedades

// y no tienen otro propósito.

internal class Bacon { }

internal class Coffee { }

internal class Egg { }

internal class Juice { }

internal class Toast { }

class Program

{

// Marcamos Main como async Task para poder usar await dentro de él.

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("el café está listo");

// Iniciar todas las tareas de cocción en paralelo inmediatamente.

// La tarea de las tostadas ahora es una composición de asincrónico y sincrónico.

var eggsTask = FryEggsAsync(2);

var baconTask = FryBaconAsync(3);

var toastTask = MakeToastWithButterAndJamAsync(2); // Usamos el nuevo método compuesto

// Servir el jugo de naranja. Esta es una operación rápida que

// puede ejecutarse mientras los demás platos se cocinan.

Juice oj = PourOJ();

Console.WriteLine("el jugo de naranja está listo");

// Ahora, esperamos los resultados de las tareas.

// No importa en qué orden se completen las tareas subyacentes,

// el await detendrá la ejecución de Main hasta que el resultado esté listo.

var eggs = await eggsTask;

Console.WriteLine("los huevos están listos");

var bacon = await baconTask;

Console.WriteLine("el tocino está listo");

var toast = await toastTask; // Aquí se esperan las tostadas (con mantequilla y mermelada)

Console.WriteLine("las tostadas están listas");

Console.WriteLine("¡El desayuno está listo!");

}

// Nuevo método para componer la tarea de tostar con el trabajo sincrónico.

// Es async Task<Toast> porque internamente usa await y devuelve un Toast.

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number); // Aquí se realiza la parte asíncrona

ApplyButter(toast); // Trabajo sincrónico

ApplyJam(toast); // Trabajo sincrónico

return toast;

}

private static Juice PourOJ()

{

Console.WriteLine("Sirviendo jugo de naranja");

return new Juice();

}

private static void ApplyJam(Toast toast) =>

Console.WriteLine("Poniendo mermelada en la tostada");

private static void ApplyButter(Toast toast) =>

Console.WriteLine("Poniendo mantequilla en la tostada");

private static async Task<Toast> ToastBreadAsync(int slices)

{

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("Poniendo una rebanada de pan en la tostadora");

}

Console.WriteLine("Comenzando a tostar...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine("Sacando las tostadas de la tostadora");

return new Toast();

}

private static async Task<Bacon> FryBaconAsync(int slices)

{

Console.WriteLine($"poniendo {slices} rebanadas de tocino en la sartén");

Console.WriteLine("cocinando el primer lado del tocino...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("volteando una rebanada de tocino");

}

Console.WriteLine("cocinando el segundo lado del tocino...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine("Poner el tocino en el plato");

return new Bacon();

}

private static async Task<Egg> FryEggsAsync(int howMany)

{

Console.WriteLine("Calentando la sartén para huevos...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine($"rompiendo {howMany} huevos");

Console.WriteLine("cocinando los huevos...");

await Task.Delay(3000); // Esperamos de forma no bloqueante

Console.WriteLine("Poner los huevos en el plato");

return new Egg();

}

private static Coffee PourCoffee()

{

Console.WriteLine("Sirviendo café");

return new Coffee();

}

}

}

Este cambio de código ilustra una técnica importante para trabajar con código asincrónico. Puedes componer tareas separando las operaciones en un método nuevo que devuelva una tarea. Puede elegir cuándo esperar para continuar con esa tarea. Puede iniciar otras tareas simultáneamente.

### Control de excepciones asincrónicas

Hasta este punto, el código supone implícitamente que todas las tareas asincrónicas se completan sin errores. Sin embargo, los métodos async pueden lanzar excepciones, al igual que sus equivalentes sincrónicos. El enfoque para manejar errores en programación asincrónica es coherente con el de la programación tradicional: escribir el código como si se tratara de una secuencia de instrucciones normales y capturar las excepciones donde corresponda.

Cuando se aplica await a una tarea que no pudo completarse correctamente, la excepción que ocurrió durante la ejecución se propaga en ese punto, lo que permite manejarla con un bloque try-catch en el código cliente.

En el ejemplo del desayuno, imaginemos que la tostadora se incendia mientras se tuesta el pan. Podemos simular este escenario modificando el método ToastBreadAsync para que genere una excepción, como se muestra a continuación:

private static async Task<Toast> ToastBreadAsync(int slices)

{

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("Poniendo una rebanada de pan en la tostadora");

}

Console.WriteLine("Comenzando a tostar...");

await Task.Delay(2000); // Espera 2 segundos

Console.WriteLine("¡Fuego! ¡La tostada está arruinada!");

// ¡Aquí es donde se lanza la excepción de la tostadora!

throw new InvalidOperationException("La tostadora está en llamas");

// Las líneas de código después de 'throw' nunca se ejecutarán.

// await Task.Delay(1000);

// Console.WriteLine("Sacando las tostadas de la tostadora");

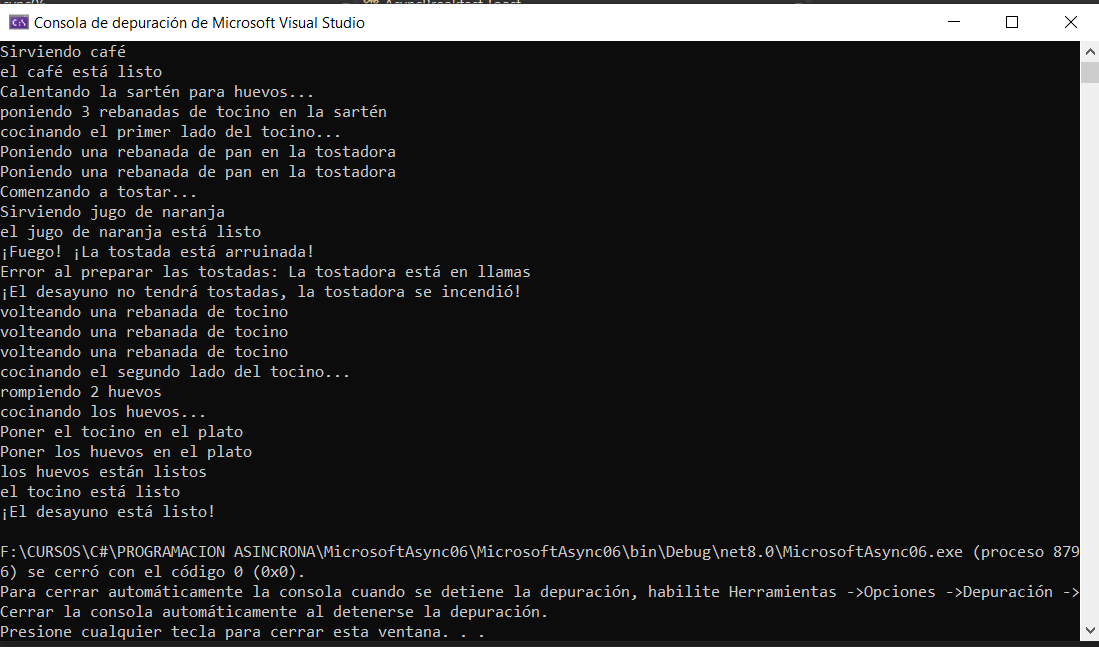
// return new Toast();

}

Al ejecutar el programa ya modificado, la salida se verá como la siguiente:

**Figura 3.**

*Salida de la ejecución del programa de ejemplo atrapando un error*



Es importante tener en cuenta que varias tareas pueden completarse exitosamente entre el momento en que se genera una excepción (como el incendio en la tostadora) y el momento en que dicha excepción es detectada por el sistema. Cuando una tarea asincrónica falla, dicha tarea se considera en estado de error y almacena la excepción generada en su propiedad Task.Exception.

Cuando se aplica una expresión await sobre una tarea que ha fallado, esta vuelve a lanzar la excepción que se produjo durante su ejecución.

Existen dos aspectos fundamentales que debes comprender sobre este mecanismo:

* Cómo se almacena una excepción en una tarea fallida.
* Cómo se desempaqueta esa excepción y se vuelve a lanzar cuando el código espera (con await) una tarea que falló.

Cuando se produce una excepción dentro de un método asincrónico, esta se encapsula en un objeto Task. La propiedad Task.Exception contiene un objeto de tipo System.AggregateException, ya que una operación asincrónica puede producir múltiples excepciones. Las excepciones individuales se almacenan dentro de la colección AggregateException.InnerExceptions.

Si la propiedad Exception es null, se crea una nueva instancia de AggregateException con la excepción capturada como su primer elemento en la colección.

En la práctica, lo más común es que la propiedad Exception contenga solo una excepción. Cuando se espera (await) una tarea fallida, se vuelve a lanzar automáticamente la primera excepción contenida en InnerExceptions. Por eso, en el ejemplo del desayuno, lo que se muestra como salida es un objeto de tipo System.InvalidOperationException y no un AggregateException. Esta forma de manejar las excepciones facilita que el código asincrónico se comporte de forma similar al código sincrónico en lo que respecta a errores.

No obstante, si anticipas múltiples errores en una operación, puedes acceder manualmente a la colección InnerExceptions para analizarlos.

**Sugerencia**

La práctica recomendada es que las excepciones por validación de argumentos se lancen de manera sincrónica, incluso en métodos que devuelven tareas.

### Nota de seguridad

Antes de continuar con la siguiente sección, asegúrate de **comentar** las siguientes dos líneas del método ToastBreadAsync, para evitar que el ejemplo vuelva a simular un incendio:

Console.WriteLine("¡Fuego! ¡La tostada está arruinada!");

// ¡Aquí es donde se lanza la excepción de la tostadora!

throw new InvalidOperationException("La tostadora está en llamas");

La serie de expresiones await al final del ejemplo anterior puede mejorarse utilizando métodos utilitarios que proporciona la clase Task. Uno de los métodos más útiles en este contexto es Task.WhenAll, que permite esperar de forma eficiente a que se completen varias tareas simultáneamente.

#### Task.WhenAll

El método WhenAll devuelve un objeto Task que se completa cuando todas las tareas especificadas en su lista de argumentos han finalizado, ya sea correctamente o con error. Esto permite manejar múltiples operaciones asincrónicas en paralelo de una manera más limpia y concisa.

A continuación, se muestra un ejemplo de cómo utilizar WhenAll para esperar la finalización de las tareas de cocción del desayuno:

await Task.WhenAll(eggsTask, baconTask, toastTask);

Este enfoque reduce el número de líneas de código y permite detectar simultáneamente las excepciones de cualquiera de las tareas si las hubiera.

#### Task.WhenAny

Otra alternativa útil es el método Task.WhenAny, que devuelve un Task<Task>, es decir, una tarea que se completa cuando **cualquiera** de las tareas especificadas finaliza. Esto es útil cuando necesitas reaccionar tan pronto como se complete la primera tarea, sin esperar a que terminen las demás.

El flujo habitual con WhenAny es:

1. Esperar la tarea devuelta por WhenAny (ya que esa tarea está garantizado que ha finalizado).
2. Procesar su resultado.
3. Eliminar la tarea completada del conjunto de tareas activas.
4. Repetir el proceso si hay más tareas pendientes.

Este patrón es especialmente útil cuando deseas mostrar resultados parciales a medida que van llegando, o si algunas tareas pueden tardar considerablemente más que otras.

var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };

while (breakfastTasks.Count > 0)

{

Task finishedTask = await Task.WhenAny(breakfastTasks);

if (finishedTask == eggsTask)

{

Console.WriteLine("Eggs are ready");

}

else if (finishedTask == baconTask)

{

Console.WriteLine("Bacon is ready");

}

else if (finishedTask == toastTask)

{

Console.WriteLine("Toast is ready");

}

await finishedTask;

breakfastTasks.Remove(finishedTask);

}

Cerca del final del fragmento de código, se puede observar la instrucción await finishedTask;. Es importante entender que la expresión await Task.WhenAny(...) no espera a que la tarea completada finalice completamente; lo que hace es devolver inmediatamente la tarea que terminó primero, ya sea de forma satisfactoria o con error.

Por esta razón, la práctica recomendada es realizar un segundo await directamente sobre la tarea retornada (finishedTask), aunque se sepa que ya ha finalizado. Esto permite:

* Obtener correctamente el resultado de la tarea (en caso de que lo retorne).
* Asegurarse de que se propague cualquier excepción que haya ocurrido durante su ejecución.

Este patrón garantiza que no se omita ningún error y que el comportamiento del código asincrónico sea coherente con el manejo esperado de excepciones y resultados.

### Versión final del código de ejemplo

Así queda la versión final:

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace AsyncBreakfast

{

// These classes are intentionally empty for the purpose of this example. They are simply marker classes for the purpose of demonstration, contain no properties, and serve no other purpose.

internal class Bacon { }

internal class Coffee { }

internal class Egg { }

internal class Juice { }

internal class Toast { }

class Program

{

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Console.WriteLine("coffee is ready");

var eggsTask = FryEggsAsync(2);

var baconTask = FryBaconAsync(3);

var toastTask = MakeToastWithButterAndJamAsync(2);

var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };

while (breakfastTasks.Count > 0)

{

Task finishedTask = await Task.WhenAny(breakfastTasks);

if (finishedTask == eggsTask)

{

Console.WriteLine("eggs are ready");

}

else if (finishedTask == baconTask)

{

Console.WriteLine("bacon is ready");

}

else if (finishedTask == toastTask)

{

Console.WriteLine("toast is ready");

}

await finishedTask;

breakfastTasks.Remove(finishedTask);

}

Juice oj = PourOJ();

Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");

}

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number);

ApplyButter(toast);

ApplyJam(toast);

return toast;

}

private static Juice PourOJ()

{

Console.WriteLine("Pouring orange juice");

return new Juice();

}

private static void ApplyJam(Toast toast) =>

Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>

Console.WriteLine("Putting butter on the toast");

private static async Task<Toast> ToastBreadAsync(int slices)

{

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("Putting a slice of bread in the toaster");

}

Console.WriteLine("Start toasting...");

await Task.Delay(3000);

Console.WriteLine("Remove toast from toaster");

return new Toast();

}

private static async Task<Bacon> FryBaconAsync(int slices)

{

Console.WriteLine($"putting {slices} slices of bacon in the pan");

Console.WriteLine("cooking first side of bacon...");

await Task.Delay(3000);

for (int slice = 0; slice < slices; slice++)

{

Console.WriteLine("flipping a slice of bacon");

}

Console.WriteLine("cooking the second side of bacon...");

await Task.Delay(3000);

Console.WriteLine("Put bacon on plate");

return new Bacon();

}

private static async Task<Egg> FryEggsAsync(int howMany)

{

Console.WriteLine("Warming the egg pan...");

await Task.Delay(3000);

Console.WriteLine($"cracking {howMany} eggs");

Console.WriteLine("cooking the eggs ...");

await Task.Delay(3000);

Console.WriteLine("Put eggs on plate");

return new Egg();

}

private static Coffee PourCoffee()

{

Console.WriteLine("Pouring coffee");

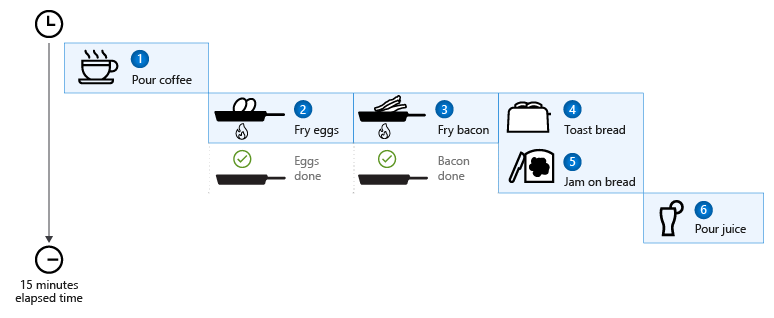
return new Coffee();

}

}

}

**Figura 4.**

*Versión final del ejemplo asincrónico*

Nota: Recuperado de: <https://learn.microsoft.com/es-es/dotnet/csharp/asynchronous-programming/>

El código completa las tareas asincrónicas del desayuno en aproximadamente 15 minutos. Este tiempo total se reduce porque varias tareas se ejecutan de forma simultánea. El programa supervisa múltiples tareas en paralelo y actúa únicamente cuando es necesario.

El resultado es un código completamente asincrónico, que refleja con mayor precisión cómo una persona organizaría la preparación de un desayuno en la vida real. Al compararlo con el primer ejemplo presentado en este artículo, se observa que las acciones principales siguen siendo fáciles de entender al leer el código. La estructura final del programa puede leerse de forma muy similar a la lista de pasos para hacer el desayuno mostrada al principio: inicia las tareas tan pronto como sea posible y evita bloquearte mientras esperas que terminen. Las palabras clave async y await permiten traducir esas instrucciones de manera natural al código, logrando un flujo más realista y eficiente.

**Conclusiones.**

La exploración detallada del Modelo de Programación Asincrónica Basado en Tareas (TAP) en C#, mediante el uso de async y await y la analogía de la preparación del desayuno, ha demostrado de manera contundente la superioridad de los enfoques asincrónicos sobre la programación sincrónica tradicional en escenarios donde existen operaciones con tiempos de espera significativos.

Se ha evidenciado que la programación sincrónica, si bien sencilla, es ineficiente en entornos modernos, llevando a bloqueos del hilo de ejecución y comprometiendo la reactividad de la interfaz de usuario y la escalabilidad del servidor. En contraste, la introducción de async y await permite desvincular la ejecución del hilo, liberándolo para otras tareas y manteniendo la aplicación responsiva, incluso si el tiempo total de la operación no se reduce inmediatamente.

La optimización de la ejecución asincrónica, lograda mediante el inicio concurrente de tareas independientes y la postergación estratégica de los await, reduce de forma sustancial el tiempo global de procesamiento. Esto subraya la capacidad del modelo TAP para maximizar la eficiencia del tiempo, permitiendo que las latencias de múltiples operaciones se superpongan eficazmente. Asimismo, la composición de tareas asincrónicas reafirma la necesidad de tratar toda una operación como asincrónica si cualquiera de sus componentes lo es, garantizando un flujo de trabajo sin bloqueos.

En definitiva, async y await no solo simplifican la escritura de código asincrónico al hacerlo más legible y secuencial, sino que también son herramientas indispensables para el desarrollo de aplicaciones robustas, de alto rendimiento y con una excelente experiencia de usuario en el ecosistema .NET. Su correcta implementación es un pilar fundamental para abordar los desafíos de concurrencia en la computación actual.

**Referencias.**

BillWagner. (2025). Programación asincrónica - C#. Retrieved from <https://learn.microsoft.com/es-es/dotnet/csharp/asynchronous-programming/>