Node, C10k y async network programming

¿De qué va esto de NodeJS y qué relación tiene con la programación asíncrona de servidores de red? ¿Y qué es eso del C10k problem?

Pues estas cosas y algunas más son las que pretendo contar y compartir en este post :-). Mi objetivo original era aclarar mis propias ideas y las de mi equipo en Plastic SCM sobre estos temas, dejando por escrito algunas notas. Pero una vez en marcha, he pensado que quizá podría ser interesante para más gente, así que allá vamos.

¿Qué es Node?

Os sonará NodeJS, la tecnología para desarrollar aplicaciones servidor en JavaScript. Supongo que hay muchas formas de definir qué es y qué ventajas tiene, pero desde mi perspectiva y centrado en escribir servidores escalables, Node es una forma de escribir software escalable basado en programación asíncrona en lugar de multi-threaded.

Simplificando mucho, al escribir código Node siempre te basas en que se va a ejecutar sobre un único thread, y no hay nunca nada que sincronizar. Eso sí, cada vez que hagas una operación de IO tu "thread" perderá el control para que otras peticiones tengan la oportunidad de procesarse. Así que cada acceso a fichero o red será una llamada *async*.

No sé mucho más sobre NodeJS, pero sí que se basa en la librería libuv que sirve para escribir servidores de red muy escalables, algo que me interesa mucho.

El C10K problem

Este C10K es una de esas cosas "que hay que saber" para realmente parecer que sabes de lo que hablas ;-) En mi opinión es el equivalente en network programming a poder citar Mythical Man Month en ingeniería de software :-)

Se trata simplemente de un término que se ha hecho popular para referirse a cómo diseñar servidores de sockets capaces de gestionar muchísimos clientes a la vez.

En la universidad hacemos nuestro primer programa de sockets que simplemente atiende una petición a la vez. Luego, al menos en mis tiempos, se pasaba a un forked server, pero muchas veces no te quedabas con la importancia de todo aquello. Aunque tras el TCP/IP Illustrated de Stevens había uno, en mi opinión mucho más interesante, que cubría casi todo: Unix Network Programming.

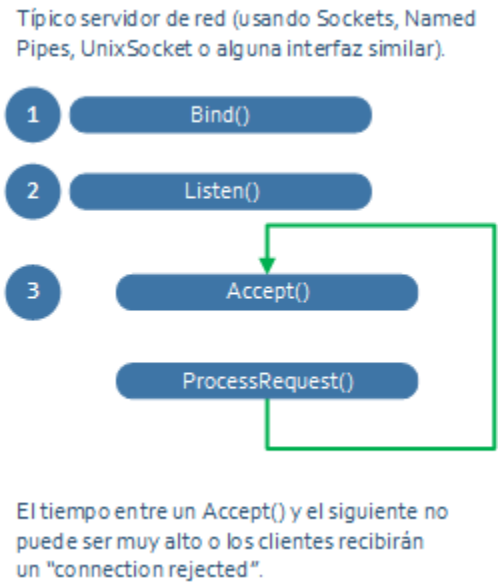
En definitiva, la página del C10K recoge unas cuantas alternativas sobre cómo construir servidores escalables. Comienza con un proceso para todo, luego va hacia los forked y multi-threaded, explora luego epoll y similares para desacoplar los sockets de los threads y después avanza hacia temas asíncronos (relacionados con lo anterior, epoll y compañía) que son la base de libuv y Node.

Es una fuente de referencia muy buena para tener en cuenta alternativas o bien saber por qué funcionan cómo funcionan ciertas tecnologías, como puede ser Node, Ngnx, etc.

El servidor de sockets más básico

Para que el post no sea sólo una colección de links (que no obstante explicarán todo esto en más detalle que yo), voy a contar algunas de las alternativas para construir servidores de red (de sockets típicamente).

Esta es la pinta de un servidor de sockets muy básico:



Haces un bind al puerto en el que quieres escuchar, luego un listen, y después un bucle en el que hay un accept para esperar una conexión del cliente (accept() devuelve un socket listo para usarse), y se procesa la petición: típicamente leer datos del socket, hacer algo, y escribir datos en el socket.

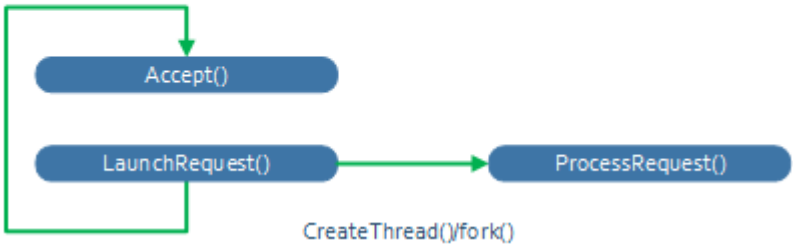
Esta estructura de servidor puede ser suficiente si el "ProcessRequest()" es súper rápido. Pero en cuanto ese procesamiento se alargue, los clientes pueden comenzar a recibir "connection rejected" si el servidor no está haciendo "accepts" como debería.

El "forked" o multi-threaded server

El siguiente paso natural es crear un servidor multi-proceso o multi-thread (ojo, no son lo mismo, pero lo cuento un poco mejor luego).

El bucle pasa a ser algo como esto:

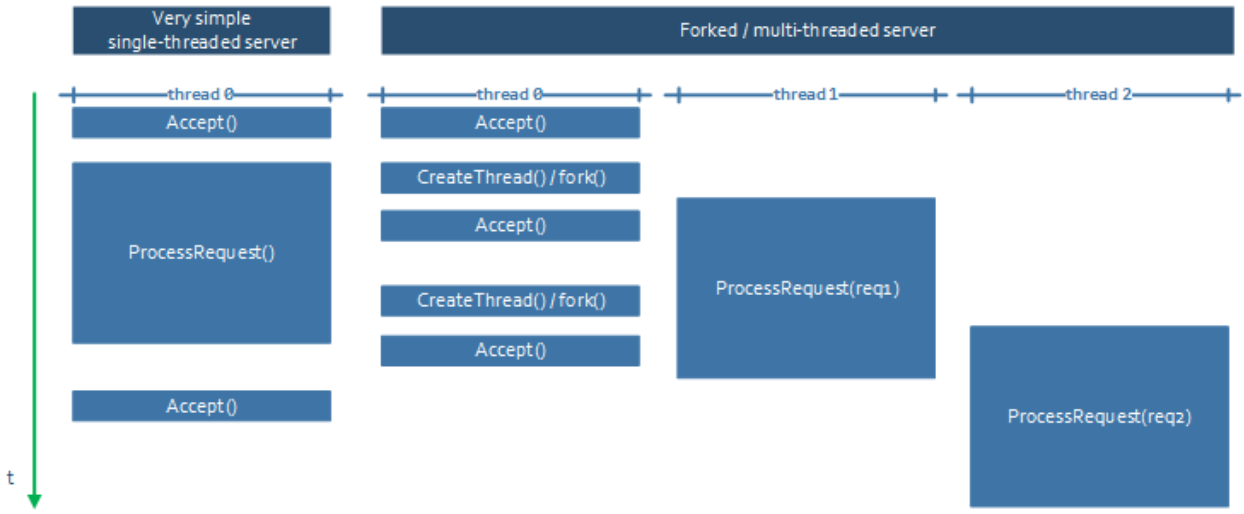
Opción 1) Nuevo thread/proceso por request



Y cuando programas esto por primera vez crees que la escalabilidad ya no tiene misterios para ti ;-)

El bucle principal puede retornar de inmediato al "accept()" de modo que ninguna nueva petición de conexión se rechazará. Y un nuevo proceso o un nuevo thread atenderá la petición, así que el sistema escalará mucho porque cualquier CPU moderna admite muchos threads (o eso creemos).

El siguiente gráfico explica un poco lo que ocurre con los diseños single-thread y "forked()":



Se ve claro que en el "single-threaded" las peticiones se procesan secuencialmente pero con este nuevo modelo el servidor puede procesar más de una en paralelo. Y además el "accept" nunca está desatendido.

Procesos y threads

He usado procesos y threads como si fueran lo mismo en el punto anterior, y por supuesto hay diferencias enormes, especialmente en ciertos sistemas operativos.

Se le suele llamar "forked server" porque en los ejemplos básicos de servidores de sockets en Unix se hacía un "fork()", crear un nuevo proceso, para atender a cada nueva petición.

Se puede afirmar que todo proceso tiene al menos un thread. El thread es lo que realmente "ejecuta" el sistema operativo (el scheduler salta de uno a otro).

Un proceso puede tener múltiples threads, y en general crear un nuevo thread es mucho más rápido que crear un nuevo proceso.

Y aquí hay diferencias entre diferentes sistemas operativos:

- En Linux durante años (esto ya es un poco viejo) crear un nuevo thread suponía crear un nuevo proceso. Pero ojo, es que crear un proceso nuevo era súper ligero. Por eso los forked servers eran viables. De hecho, el servidor Apache fue "forked" durante años (esto ya es arqueología de software). Los pthreads mapeaban un proceso "hijo" a cada thread. Esto cambió ya hace mucho y aunque los procesos siguen siendo ligeros en comparación con otros OS, un proceso puede gestionar múltiples threads de forma sencilla y eficiente.
- En Windows un proceso se crea con CreateProcess (o cuando tu programa arranca, que ya es un proceso) y es algo pesado. Sin embargo, crear un thread es algo súper rápido. Un "forked server" no sería viable en Windows, pero sí un multi-threaded server.

La separación entre procesos es mucho más fuerte que entre threads. Los threads comparten memoria mientras que los procesos no (a menos que se haga "a propósito" con memory mapped files, pero eso es otra historia), por lo que en ocasiones tener procesos diferentes hace la solución más robusta frente a crashes.

La implementación de los threads también es diferente en distintos sistemas operativos: en Windows y Linux un user-mode thread (los que el programador crea) se mapean con un kernel thread. En Solaris (que en su día tuvo peso :P) se jactaban de que era mucho mejor su solución de varios user mode threads mapeados sobre 1 kernel thread. Sin embargo en Solaris 9 (o 10, ya no me acuerdo) pasaron al mismo mapeo 1-1 de Windows. (Hace años solía leer sobre estos temas, y había unos blogposts y un libro sobre Solaris muy, muy interesante: Solaris Internals, que junto a Windows Internals y Linux Systems Programming te dan una visión muy sólida, a la que yo al menos he sacado partido durante años).

Curiosamente vamos a ver cómo ahora con async programming se vuelve a gestionar "threads lógicos" sobre "1 thread físico", así que la moda va y vuelve.

Por último, crear threads, aunque rápido, no es gratis. Cada thread suele ocupar 1MB de memoria de kernel, así que un servidor súper ocupado con 1000 clientes a la vez, tendría 1GB de RAM usado solamente gestionando la infraestructura más básica de threads (además que mil threads estarían gastando casi más tiempo en pasar de uno a otro que en hacer nada de trabajo real).

Desacoplar sockets y threads

Hasta ahora, tanto en single-thread como multi-thread, un thread atiende a un único socket.

El código puede ser algo como:

```
void ProcessRequest(Socket s)
{
      do
      {
            data = ReadFromSocket(s);
            response = Process(data);
```

```
            SendReponse(s);
      }while (socket not closed)
}
```

Es decir, lo normal es que el cliente abra una conexión y utilice esa misma conexión para realizar más de una petición (request) porque si no cerrar y reabrir el socket cada vez es lento. Los clientes suelen cachear esas conexiones abiertas de alguna forma para no tener que reabrir cada vez.

Pero, claro, eso quiere decir que nuestro pseudocódigo de arriba vincula un thread del servidor con un cliente hasta que al cliente le dé por cerrar la conexión. Vale que una vez termine el thread, que podría estar en un "pool" podría reutilizarse, pero hay una vinculación 1 a 1, y además el thread de servidor puede estar parado esperando a que el cliente envíe una nueva request, sin hacer nada, mientras el cliente interactúa con el usuario, etc, etc.

Así que el siguiente paso en búsqueda de escalabilidad es desacoplar threads de sockets. Y ahí es donde entra en juego la llamada Select. El pseudocódigo pasaría a ser algo como:

Thread para atender a todos los clientes (1 para todos los sockets):

```
while (true)
{
      Select(socketArray[]);
      // si sale de aquí es que en alguno de
      // los sockets ha pasado algo
      Process(socketSelected);
}
```

Thread de proceso, parecido al de antes (pero habrá "n" fijos para atender a todos):

```
            data = ReadFromSocket(s);
            response = Process(data);
            SendReponse(s);
            Return the socket "s" to the "Select" somehow
```

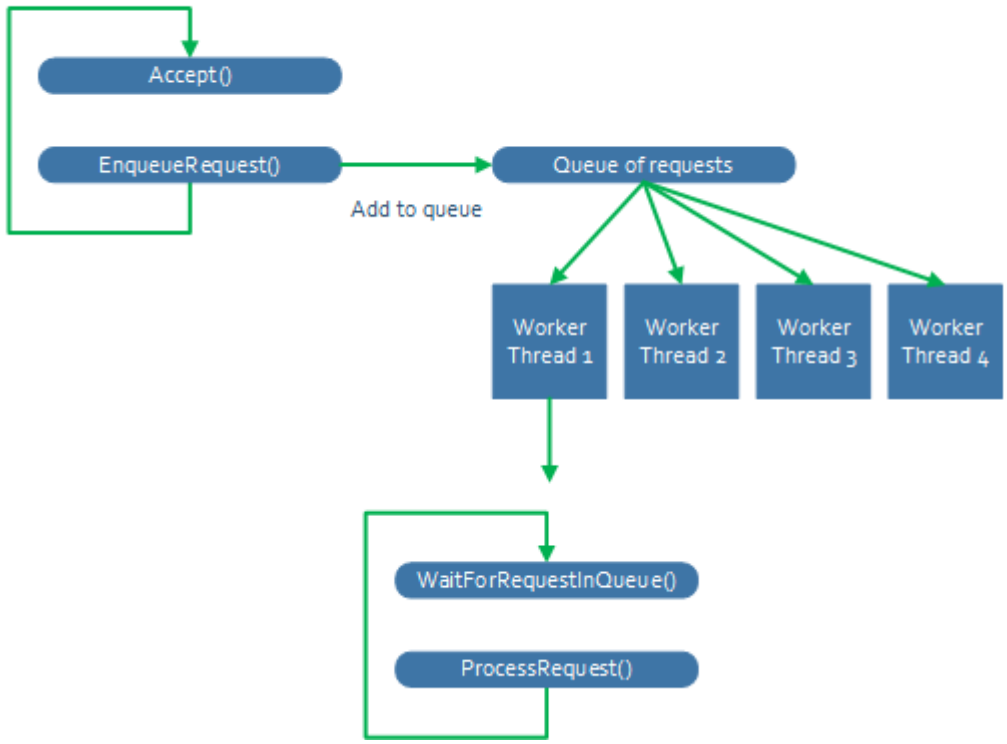
Al final en lugar de Select, que es muy antiguo y no escala bien para centenares de handles de socket, se suele usar epoll en Linux, o Completion Ports en Windows, o algo como libuv que se construye por encima y lo hace más fácil. O las llamadas asíncronas en C# que hacen justo eso en Linux (Mono) y Windows (.NET) y supongo que lo mismo en Java.

Cola de peticiones + thread pool

Al describir cómo desacoplar sockets de threads surge otro concepto que es el de encolar las peticiones y atenderlas con un thread pool con un número controlado de threads.

El esquema pasa a ser algo como esto:

Opción 2) Cola de peticiones + Thread Pool



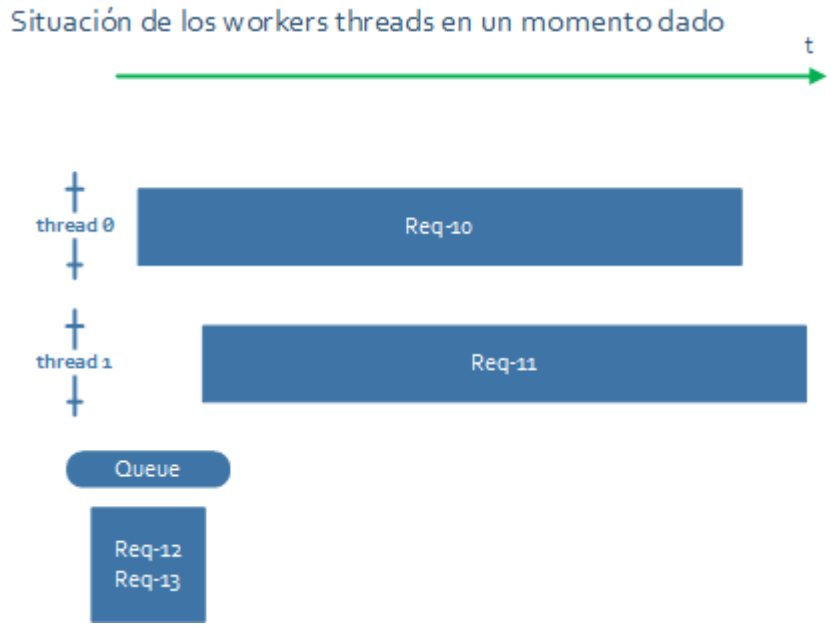
El bucle de "accept" solamente encola, y luego hay un número de "workers" que atienden las peticiones. Ojo, esto hay que combinarlo con lo que hemos visto de desacoplar threads de sockets, de modo que en algún punto se estará haciendo un "Select" o equivalente escalable cada vez que se espere por nuevos datos de una conexión abierta, y esas conexiones listas para ser procesadas se meterán también en esa cola de peticiones.

Al final, en un servidor con 24 cores, por poner un ejemplo, es mejor opción tener 24 threads atendiendo peticiones, y que las que no estén atendidas estén encoladas, que crear un número ilimitado de threads. Se usa mejor la CPU y se evita que el servidor "muera" a base de tener tantos hilos que prácticamente ninguno puede avanzar.

Async – o cómo aprovechar mejor los threads

Vamos a ver qué pasa cuando tenemos ya montado un sistema con varios threads para atender peticiones que se van encolando. Para mantenerlo sencillo, vamos a manejar solamente 2 worker threads en el pool.

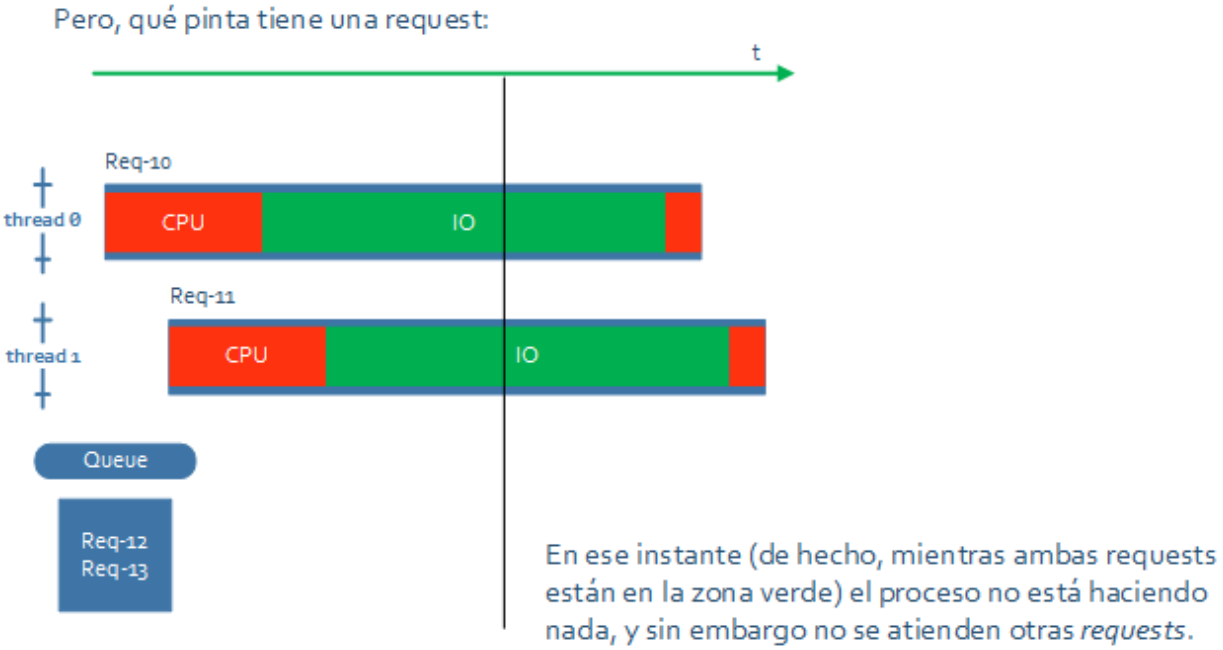
En un momento dado la situación puede ser la siguiente:



Hay 2 peticiones siendo procesadas, y otras dos pendientes en la cola.

Pero, vamos a ver qué pinta tienen las peticiones. Porque, si todo el tiempo de proceso es de CPU, efectivamente la solución ya sería suficiente, porque no habría nada más que paralelizar. Pero lo normal es que para atender una petición haya que hacer algo de entrada salida.

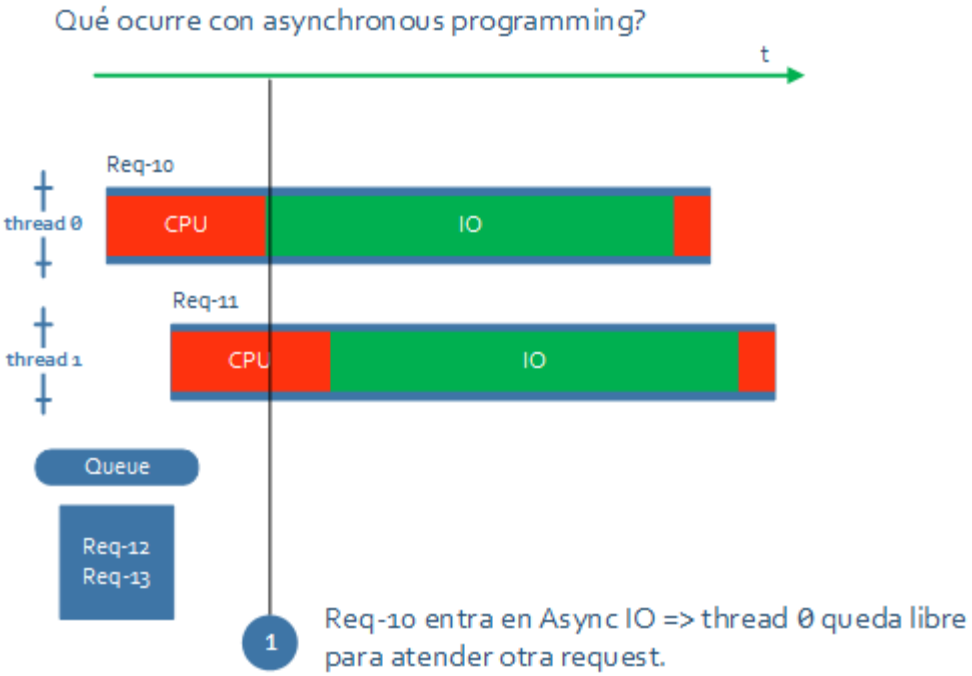
En el siguiente gráfico se ve que cada una de las *requests* del ejemplo tiene primero trabajo de CPU, luego un rato de IO, y al final otro poco de CPU.



En el instante que marca el gráfico, si los dos hilos están haciendo operaciones de IO bloqueantes, el proceso estará a 0% de CPU pero habrá requests encoladas ... Es decir, que hay CPU suficiente para hacer más cosas, hay trabajo pendiente, pero el servidor no avanza...

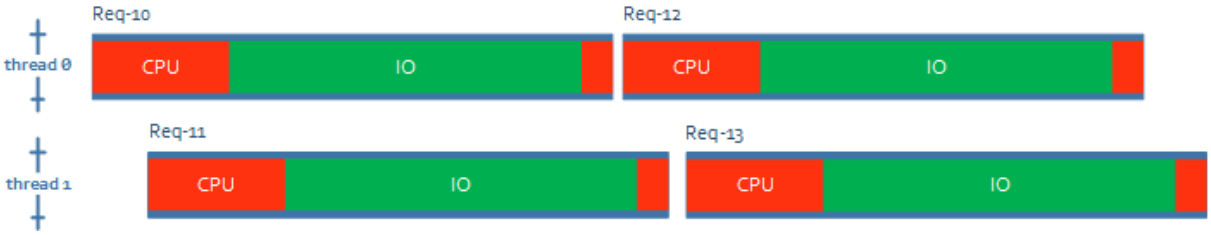
¿Se puede mejorar?

Con async IO los threads pasarán a utilizarse para procesar la parte de CPU de la request en lugar de estar esperando sin hacer nada.



De este modo se conseguirá un mejor uso de la CPU y un servidor que admita más concurrencia sin tener que usar más threads.

Mientras que sin "async" el total de la resolución sería algo como esto:



De alguna forma estas librerías async, como puede ser el framework Node, o en el caso de las pruebas que estoy haciendo para este post C#/.NET con async, lo que hacen es "scheduling en modo usuario". Es como si hubiera "threads sobre threads". En Windows hubo un intento de resolver esto hace muchos años con algo que llamaban Fibers, pero que creo que no es lo que usan al final todas estas librerías.

En C#/.NET lo que se usa es el concepto de "futures" y "continuations" en el que se captura el estado del código (parecido a todo lo de las lambdas, etc) y se puede volver a ese estado más adelante.

Básicamente, cada vez que una request entra en IO (o en cualquier operación asíncrona), su thread se libera para continuar avanzando con otra request, y cuando esta entre en IO, la original continuará.

¿No podría hacer ya todo eso el sistema operativo con los threads sin tener que complicarse la vida en modo usuario? Pues, podría ser, pero la realidad es que hoy por hoy, esta forma de gestionar la concurrencia parece más óptima y que da mejores resultados.

Ejemplos de código

He preparado un pequeño ejemplo en C# para ilustrar el comportamiento async. No hay sockets ni uso de CPU, pero simulo el comportamiento con un "sleep" síncrono (uso de CPU) y uno asíncrono (que sería el IO).

El código se puede encontrar en https://github.com/psantosl/asyncexample.

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace TestAsyncProcessing
{
    class Program
    {
```

```
static void Main(string[] args)
{
    ThreadPool.SetMaxThreads(1, 1);

    for (int i = 0; i < 4; ++i)
    {
        string name = i.ToString();
        Task.Factory.StartNew(() =>
        { new Request("req" + name).ProcessRequest(); });
    }

    Console.WriteLine(
        "Type a name and then ENTER to schedule a new request");
    string order = Console.ReadLine();

    while (true)
    {

        Task.Factory.StartNew(() =>
            {
                Request t = new Request(order);

                t.ProcessRequest();
            }
        );
    }
}

class Request
{
    internal Request(string name)
    {
        mName = name;
    }

    internal async void ProcessRequest()
    {
        int ini = Environment.TickCount;

        WriteLine(mName, "Starting request");

        WriteLine(mName, "Do heavy calculation ...");

        Thread.Sleep(1000);

        WriteLine(mName, "Done. {0} ms",
            Environment.TickCount - ini);

        WriteLine(mName, "Sleeping to do async");

        // replace await by Thread.Sleep
        // to simulate blocking vs async IO
        await Task.Delay(10000);

        //Thread.Sleep(10000);

        WriteLine(mName,
            "Big async operation. Request terminated."+
            " {0} ms since start to complete",
            Environment.TickCount - mStart);
    }
```

```
            void WriteLine(
                string requestName,
                string format,
                params object[] args)
            {
                string s = string.Format(format, args);

                Console.WriteLine(
                    "{0} - thId - {1} - {2}",
                    requestName,
                    Thread.CurrentThread.ManagedThreadId, s);
            }

            string mName;
        }

        static int mStart = Environment.TickCount;
    }
}
```

Lo he ejecutado en una máquina virtual configurada con un único core, de otro modo la siguiente llamada no hará nada porque el mínimo es el número de cores del sistema.

```
            ThreadPool.SetMaxThreads(1, 1);
```

Y estos son los resultados en modo "async":

```
C:\Users\pablo\Desktop>TestAsyncProcessing.exe
req0 - thId - 3 - Starting request
req0 - thId - 3 - Do heavy calculation ...
req0 - thId - 3 - Done. 1047 ms
req0 - thId - 3 - Sleeping to do async
req1 - thId - 3 - Starting request
req1 - thId - 3 - Do heavy calculation ...
req1 - thId - 3 - Done. 1031 ms
req1 - thId - 3 - Sleeping to do async
req2 - thId - 3 - Starting request
req2 - thId - 3 - Do heavy calculation ...
req2 - thId - 3 - Done. 1016 ms
req2 - thId - 3 - Sleeping to do async
req3 - thId - 3 - Starting request
req3 - thId - 3 - Do heavy calculation ...
req3 - thId - 3 - Done. 1015 ms
req3 - thId - 3 - Sleeping to do async
req0 - thId - 3 - Big async operation. Request terminated. 11125 ms since start to complete
req1 - thId - 3 - Big async operation. Request terminated. 12141 ms since start to complete
req2 - thId - 3 - Big async operation. Request terminated. 13156 ms since start to complete
req3 - thId - 3 - Big async operation. Request terminated. 14187 ms since start to complete
```

Se ve cómo un único thread "thId 3" procesa todas las peticiones y como terminan todas en poco más de 14 segundos.

Sin embargo, ¿qué pasa cuando reemplazamos await Task.Delay(10000) por Thread.Sleep(10000) para simular una operación de IO síncrona que "bloquea" el thread?

```
C:\Users\pablo\Desktop>TestAsyncProcessing.exe
req0 - thId - 3 - Starting request
Type a name and then ENTER to schedule a new request
req0 - thId - 3 - Do heavy calculation ...
req0 - thId - 3 - Done. 1031 ms
req0 - thId - 3 - Sleeping to do async
req0 - thId - 3 - Big async operation. Request terminated. 11063 ms since start to complete
req1 - thId - 3 - Starting request
req1 - thId - 3 - Do heavy calculation ...
req1 - thId - 3 - Done. 1016 ms
req1 - thId - 3 - Sleeping to do async
req1 - thId - 3 - Big async operation. Request terminated. 22094 ms since start to complete
```

```
req2 - thId - 3 - Starting request
req2 - thId - 3 - Do heavy calculation ...
req2 - thId - 3 - Done. 1031 ms
req2 - thId - 3 - Sleeping to do async
req2 - thId - 3 - Big async operation. Request terminated. 33141 ms since start to complete
req3 - thId - 3 - Starting request
req3 - thId - 3 - Do heavy calculation ...
req3 - thId - 3 - Done. 1016 ms
req3 - thId - 3 - Sleeping to do async
req3 - thId - 3 - Big async operation. Request terminated. 44172 ms since start to complete
```

El ejemplo es un poco exagerado, porque usa 1 único thread para todo, pero creo que ilustra bastante bien lo que puede ocurrir en un sistema multi-threaded. El tiempo final de la última request pasa de procesarse en 14 segundos a 44.