Synchronous I/O antipattern

06/05/2017 • 6 minutes to read • Contributors

In this article

Blocking the calling thread while I/O completes can reduce performance and affect vertical scalability.

Problem description

A synchronous I/O operation blocks the calling thread while the I/O completes. The calling thread enters a wait state and is unable to perform useful work during this interval, wasting processing resources.

Common examples of I/O include:

- Retrieving or persisting data to a database or any type of persistent storage.
- Sending a request to a web service.
- Posting a message or retrieving a message from a queue.
- Writing to or reading from a local file.

This antipattern typically occurs because:

- It appears to be the most intuitive way to perform an operation.
- The application requires a response from a request.
- The application uses a library that only provides synchronous methods for I/O.
- An external library performs synchronous I/O operations internally. A single synchronous I/O call can block an entire call chain.

The following code uploads a file to Azure blob storage. There are two places where the code blocks waiting for synchronous I/O, the `CreateIfNotExists` method and the `UploadFromStream` method.

```
var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference("uploadedfiles");

container.CreateIfNotExists();
var blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = File.OpenRead(HostingEnvironment.MapPath("~/FileToUpload.txt")))
{
    blockBlob.UploadFromStream(fileStream);
}
```

Here's an example of waiting for a response from an external service. The `GetUserProfile` method calls a remote service that returns a `UserProfile`.

```
public interface IUserProfileService
{
    UserProfile GetUserProfile();
}

public class SyncController : ApiController
{
    private readonly IUserProfileService _userProfileService;

    public SyncController()
    {
        _userProfileService = new FakeUserProfileService();
    }

    // This is a synchronous method that calls the synchronous GetUserProfile method.
    public UserProfile GetUserProfile()
    {
        return _userProfileService.GetUserProfile();
    }
}
```

You can find the complete code for both of these examples here.

How to fix the problem

Replace synchronous I/O operations with asynchronous operations. This frees the current thread to continue performing meaningful work rather than blocking, and helps improve the utilization of compute resources. Performing I/O asynchronously is particularly efficient for handling an unexpected surge in requests from client applications.

Many libraries provide both synchronous and asynchronous versions of methods. Whenever possible, use the asynchronous versions. Here is the asynchronous version of the previous example that uploads a file to Azure blob storage.

```
var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference("uploadedfiles");

await container.CreateIfNotExistsAsync();

var blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = File.OpenRead(HostingEnvironment.MapPath("~/FileToUpload.txt")))
{
    await blockBlob.UploadFromStreamAsync(fileStream);
}
```

The `await` operator returns control to the calling environment while the asynchronous operation is performed. The code after this statement acts as a continuation that runs when the asynchronous operation has completed.

A well designed service should also provide asynchronous operations. Here is an asynchronous version of the web service that returns user profiles. The `GetUserProfileAsync` method depends on having an asynchronous version of the User Profile service.

```
public interface IUserProfileService
{
    Task<UserProfile> GetUserProfileAsync();
}
```

```
public class AsyncController : ApiController
{
    private readonly IUserProfileService _userProfileService;

    public AsyncController()
    {
        _userProfileService = new FakeUserProfileService();
    }

    // This is an synchronous method that calls the Task based GetUserProfileAsync method.
    public Task<UserProfile> GetUserProfileAsync()
    {
        return _userProfileService.GetUserProfileAsync();
    }
}
```

For libraries that don't provide asynchronous versions of operations, it may be possible to create asynchronous wrappers around selected synchronous methods. Follow this approach with caution. While it may improve responsiveness on the thread that invokes the asynchronous wrapper, it actually consumes more resources. An extra thread may be created, and there is overhead associated with synchronizing the work done by this thread. Some tradeoffs are discussed in this blog post: Should I expose asynchronous wrappers for synchronous methods?

Here is an example of an asynchronous wrapper around a synchronous method.

```
// Asynchronous wrapper around synchronous library method
private async Task<int> LibraryIOOperationAsync()
{
    return await Task.Run(() => LibraryIOOperation());
}
```

Now the calling code can await on the wrapper:

```
// Invoke the asynchronous wrapper using a task
await LibraryIOOperationAsync();
```

Considerations

- I/O operations that are expected to be very short lived and are unlikely to cause contention might be more performant as synchronous operations. An example might be reading small files on an SSD drive. The overhead of dispatching a task to another thread, and synchronizing with that thread when the task completes, might outweigh the benefits of asynchronous I/O. However, these cases are relatively rare, and most I/O operations should be done asynchronously.

- Improving I/O performance may cause other parts of the system to become bottlenecks. For example, unblocking threads might result in a higher volume of concurrent requests to shared resources, leading in turn to resource starvation or throttling. If that becomes a problem, you might need to scale out the number of web servers or partition data stores to reduce contention.

How to detect the problem

For users, the application may seem unresponsive or appear to hang periodically. The application might fail with timeout exceptions. These failures could also return HTTP 500 (Internal Server) errors. On the server, incoming client requests might be blocked until a thread becomes available, resulting in excessive request queue lengths, manifested as HTTP 503 (Service Unavailable) errors.

You can perform the following steps to help identify the problem:

1. Monitor the production system and determine whether blocked worker threads are constraining throughput.
2. If requests are being blocked due to lack of threads, review the application to determine which operations may be performing I/O synchronously.
3. Perform controlled load testing of each operation that is performing synchronous I/O, to find out whether those operations are affecting system performance.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Monitor web server performance

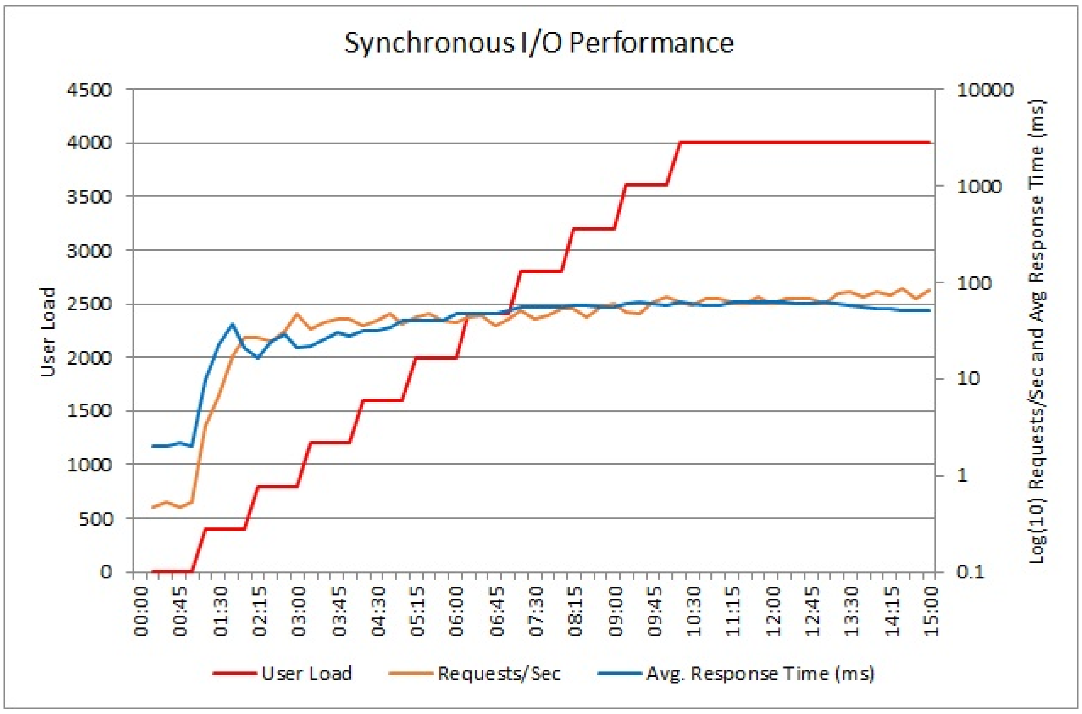
For Azure web applications and web roles, it's worth monitoring the performance of the IIS web server. In particular, pay attention to the request queue length to establish whether requests are being blocked waiting for available threads during periods of high activity. You can gather this information by enabling Azure diagnostics. For more information, see:

- Monitor Apps in Azure App Service
- Create and use performance counters in an Azure application

Instrument the application to see how requests are handled once they have been accepted. Tracing the flow of a request can help to identify whether it is performing slow-running calls and blocking the current thread. Thread profiling can also highlight requests that are being blocked.

Load test the application

The following graph shows the performance of the synchronous `GetUserProfile` method shown earlier, under varying loads of up to 4000 concurrent users. The application is an ASP.NET application running in an Azure Cloud Service web role.



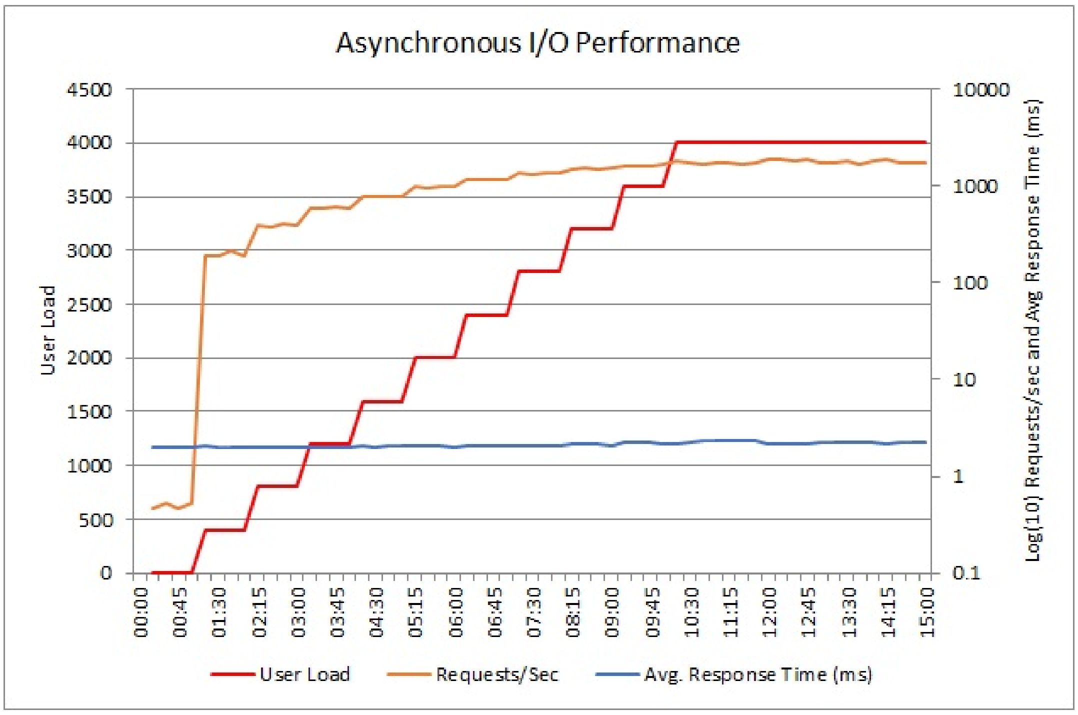
The synchronous operation is hard-coded to sleep for 2 seconds, to simulate synchronous I/O, so the minimum response time is slightly over 2 seconds. When the load reaches approximately 2500 concurrent users, the average response time reaches a plateau, although the volume of requests per second continues to increase. Note that the scale for these two measures is logarithmic. The number of requests per second doubles between this point and the end of the test.

In isolation, it's not necessarily clear from this test whether the synchronous I/O is a problem. Under heavier load, the application may reach a tipping point where the web server can no longer process requests in a timely manner, causing client applications to receive time-out exceptions.

Incoming requests are queued by the IIS web server and handed to a thread running in the ASP.NET thread pool. Because each operation performs I/O synchronously, the thread is blocked until the operation completes. As the workload increases, eventually all of the ASP.NET threads in the thread pool are allocated and blocked. At that point, any further incoming requests must wait in the queue for an available thread. As the queue length grows, requests start to time out.

Implement the solution and verify the result

The next graph shows the results from load testing the asynchronous version of the code.



Throughput is far higher. Over the same duration as the previous test, the system successfully handles a nearly tenfold increase in throughput, as measured in requests per second. Moreover, the average response time is relatively constant and remains approximately 25 times smaller than the previous test.