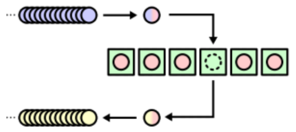
Parallel page processing with Asp.Net

Ronnie Holm



Overview

- Page rendering slows down when the thread rendering the page spends too much of its time waiting
- Explicit use of threads is cumbersome, requires quite a bit of code, and is error-prone
- Asp.Net 2.0+ has build-in support for asynchronous page processing

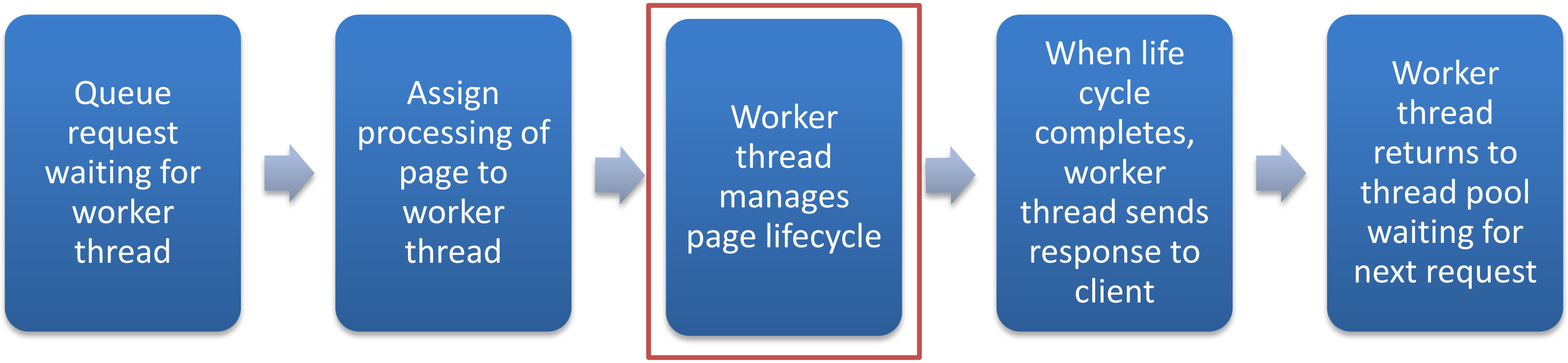
Inside w3wp.exe

- Process maintains a thread pool for servicing incoming requests
- Machine.config defines default pool setup

```
<system.web>
  <processModel autoConfig="true"/>
  <!--<processModel maxWorkerThreads="20" .../>-->
</system.web>
```

- autoConfig means Asp.Net determines the value for maxWorkerThreads based on hardware configuration (# of CPUs, cores, etc.)

Page processing overview

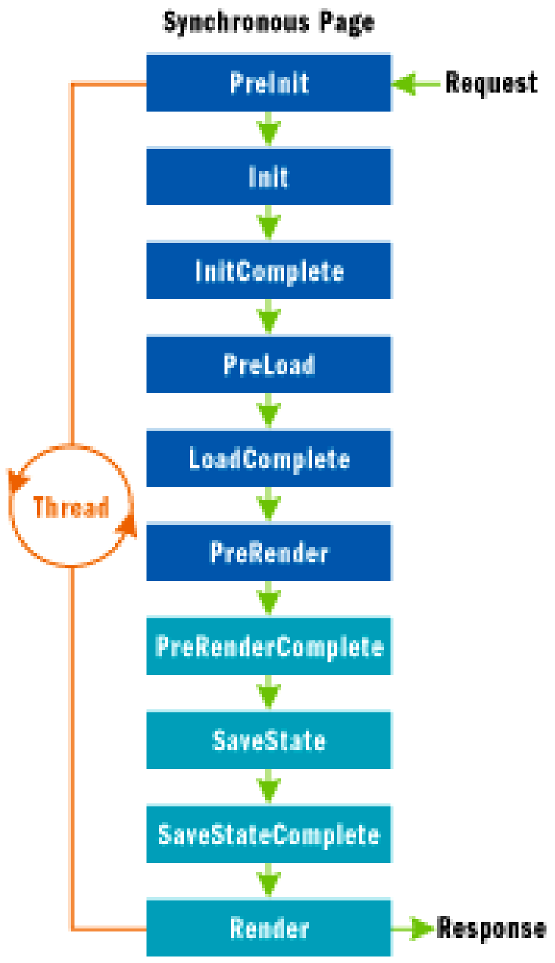


Queue request waiting for worker thread
Assign processing of page to worker thread
Worker thread manages page lifecycle
When life cycle completes, worker thread sends response to client
Worker thread returns to thread pool waiting for next request

- Too many long-running worker threads deplete the thread pool. Future requests are queued, making the site appear slow

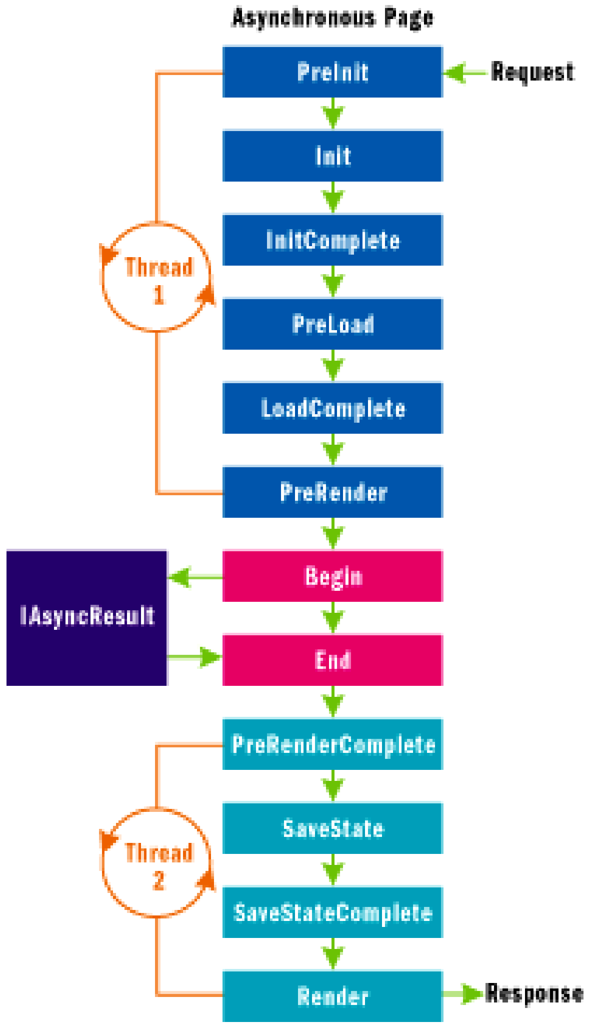
Synchronous page processing

- One worker thread responsible for entire page lifecycle
- Control code get executed on the same worker thread as page
- Worker thread may spent much of its time waiting
- Event handlers of controls gets called in the same order as event handlers of page



Asynchronous page processing

- Page/control executes code within Begin/End on separate thread
- A worker threat continues rendering other controls on page
- Page is output to client when all async calls has returned and rendering is complete
- Multiple worker threads => faster page rendering



Example

Default.aspx

```
<%@ Page Async="true"
```

Default.aspx.cs

```
public partial class Default : Page {
    protected void Page_Load(object sender, EventArgs e) {
        for (int i = 0; i < 5; i++)
            Controls.Add(new WaitControl());
    }
}
```

Debugger output

```
Page_Load: 10
Page_Load: 10
Page_Load: 10
Page_Load: 10
Page_Load: 10
DoWork: 4
DoWork: 10
DoWork: 9
DoWork: 8
DoWork: 11
Render: 11 15:16:13 15:16:18
Render: 11 15:16:13 15:16:18
Render: 11 15:16:13 15:16:18
Render: 11 15:16:13 15:16:18
Render: 11 15:16:13 15:16:19
```

WaitControl.ascx.cs

```
public partial class WaitControl : UserControl {
    private delegate void AsyncTaskDelegate();
    private AsyncTaskDelegate _task;

    // state shared between threads
    private DateTime _start, _finish;
```

1

```
    protected void Page_Load(object sender, EventArgs e) {
        Debug.WriteLine("Page_Load: " + Thread.CurrentThread.ManagedThreadId);
        var task = new PageAsyncTask(BeginAsync, EndAsync, null, null, true);
        Page.RegisterAsyncTask(task);
    }
```

2

```
    private IAsyncResult BeginAsync(object src, EventArgs args,
                                    AsyncCallback callback, object data) {
        _start = DateTime.Now;
        _task = new AsyncTaskDelegate(DoWork);
        return _task.BeginInvoke(callback, data);
    }
```

3

```
    private void DoWork() {
        Debug.WriteLine("DoWork: " + Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(5000);
    }
```

4

```
    private void EndAsync(IAsyncResult result) {
        _finish = DateTime.Now;
    }
```

5

```
    protected override void Render(HtmlTextWriter writer) {
        Debug.WriteLine("Render: " +
            Thread.CurrentThread.ManagedThreadId + " " +
            _start.ToLongTimeString() + " " +
            _finish.ToLongTimeString());
    }
}
```

Conclusion

- With little effort asynchronous processing can speed up page rendering
- Make judicious use of asynchronous processing
- Optimizing away one bottleneck most likely makes another one appear elsewhere