Server Architecture

History, Patterns and low level APIs

By Daniel Frederico Lins Leite

SERVER ARCHITECTURE

HISTORY, PATTERNS AND LOW LEVEL APIS

v0.1 beta – 2016/01/20

Author: Daniel Frederico Lins Leite (1985.daniel@gmail.com)

last version at: http://1drv.ms/1SwPIUS

Book Cover

Bridges generate passion in many people. Some consider them the ultimate engineering achievement. Some people just love them because they are beautiful. A big part of the Atlas Shrugged novel is built around the building of a bridge (ops! spoiler alert!!). However, just the minority really understand how bridges work and how to build bridges.

This book tries to explain how server middleware are built. I consider server middleware beautiful engineering achievements. Software engineering, for sure, but still beautiful engineering achievements. Therefore, at least for me, middleware is just like a bridge.

The cover shows the underground of a bridge because I expect to show you the underground of middleware.

And I sincerely hope that you like!

SUMMARY

INTRODUCTION

The .NET ecosystem is in the midst of a paradigm shift regarding its middleware. Although some of the ASPNET team decisions are not easily understandable to those of us that do not have more inside information, we can hope that these decisions will bring the .NET community to a much more mature position.

To be honest, I do not think that the default ASPNET middleware needs replacement. I am one of the few developers that still think that Webforms, for example, have its position. Sure, it could use an upgrade, version 2.0, for example, but its abandonment is a big mistake in my opinion. Please, I do not want to be misinterpreted here. I firmly believe that middleware must evolve; it must use all modern APIs and always delivers more value. I just believe that this must happens without affecting the applications on top of the middleware, because…well... That is one of the reasons for middleware to exist.

So, before jumping in the new aspects of the new middleware that ASPNET will use, let me start with a little of history.

In 1994, was released one of the major works about computer programming: the famous Gang of Four book started a tradition. It successfully standardized and named some design patterns using object-oriented approaches, and some believe that all patterns that appeared after are just little modification of the GOF patterns.

The idea proved to be so useful that after them many other works have emerged, for example: Patterns Language of Program Design, PLoPD; Pattern-Oriented System Architecture, POSA; Enterprise Integration Patterns, EIP; Patterns of Enterprise Application Architecture, P of EAA; and many others...

Off course, we cannot forget other contributions in this area, such as Ward and Cunningham paper. Especially because one of the books that this article will talk about, organize its patterns using the CRC model.

A Laboratory For Teaching Object-Oriented Thinking

http://www.inf.ed.ac.uk/teaching/courses/seoc/2007_2008/resources/CRC_OOthinking.pdf

For a more complete (and precise) history of patterns, please see http://c2.com/cgi/wiki?HistoryOfPatterns and read the first chapter of the patterns books. Each one come with its own little history about patterns.

It is a shame that the fast-paced nature of the IT world makes very easy to today developers to ignore the classics and to focus excessively only in the last buzz. For example, it is very common for today developers to totally ignore these very important books and articles because they are some years old.

My intention with this article is to incentive all developers to read the classics. Experienced knowledge must transcend the mutable and go towards the immutable. We must focus on the last frameworks and libraries to do our job; must we must see through them the classical aspects of Computer Science. To give a more cultural view to this thought I really like a quote from Saint Augustine of Hippo:

> In quorum consideratione non vana et peritura curiositas exercenda est,
> sed gradus ad immortalia et semper manentia faciendus.
>
> In the study of the being, one does not exercise an empty and futile
> curiosity; but ascend towards immortal and abiding.

Given that we are talking about middleware architecture for network systems, I would like to change the subject to the POSA Series. More specifically to the POSA II book.

POSA is a five volume book series where each book has more than 500 pages. The series describes the patterns using the CRC card (see the paper cited above), give examples and implementation guide and the authors try to enumerate all the good the bad aspects that the pattern will deliver. As the Ward and Cunningham paper say, one can become a much better developer just learning about the framework of analyzing the solution using the POSA framework.

POSA I starts with its own little history about patterns and then categorizes the patterns in three groups: Architectural Patterns, Design Patterns and Idioms. Architectural Patterns describes the “system-wide structural properties of an application”; Design Patterns describes ways of “decomposing more complex services or components”; and, Idioms are “language-specific techniques to implement particular aspects of components or its relationships”.

Therefore, to understand the current trend on server middleware I think that it is very useful to start studying two POSA patterns: the Proactor and the Reactor.

These patterns are two possible ways to architecture system that handle events. One may think: why do I have to care to events?

Well... this is a relevant question.

To answer this question, I think that I have to go back the 1990ties and try to summarize the discussion about how to architect servers.

SERVER MIDDLEWARE ARCHITECTURE EVOLUTION

This question will send us back to the decade 1990. In the decade, 1990 was an intense search towards the best architecture IO based servers.

1.1 MULTI PROCESS PARADIGM

The first paradigm was the Multi-Process, used by the Apache server on UNIX servers. Given the "cost" of process, this process has a limitation of handling dozens of requests per machine. Once a good number, but very unrealistic today event to the simplest intranet services.

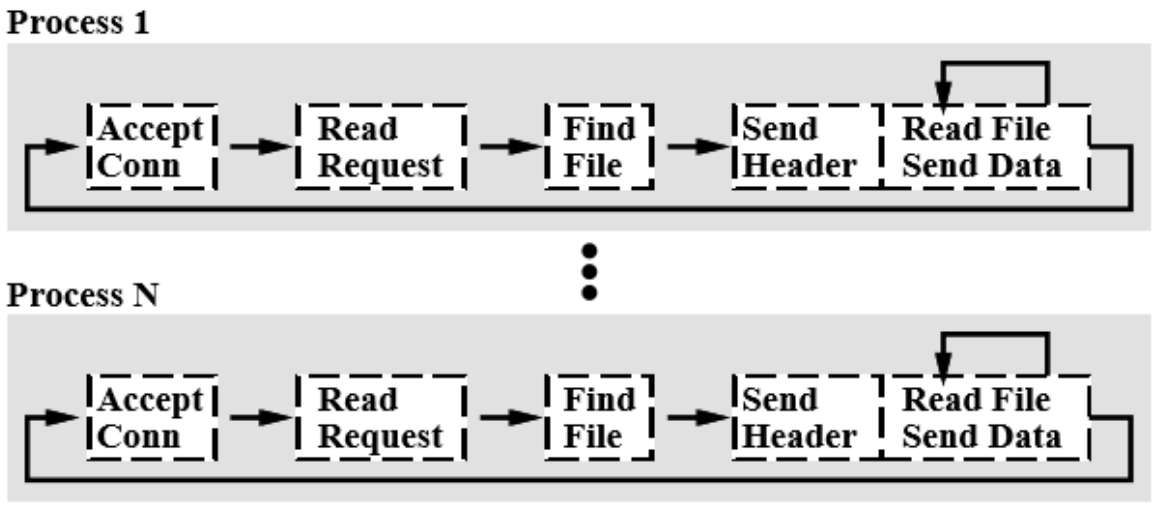


Figure 2: Multi-Process - In the MP model, each server process handles one request at a time. Processes execute the processing stages sequentially.

1.2 - MULTI THREAD PARADIGM

The natural evolution was to use one thread per request, all on the same machine. One process can easily have hundreds, even thousands of threads. What brings the capacity of the server to much more realistic numbers, especially given today machines.

The main problem with the multi-thread paradigm is that given that all threads are in the same process/address, it is necessary to use some kind of synchronization to shared resources, a problem that does not exists in the Multi-Process paradigm. Other problem is that now with a high quantity of threads a lot of processing time is wasted in thread context switch. In another words, the throughput will drastically decrease with the number of concurrent requests in this paradigm. This is why all Thread-based server have some kind of concurrency limit. The requests stay queued when the limit is achieved.

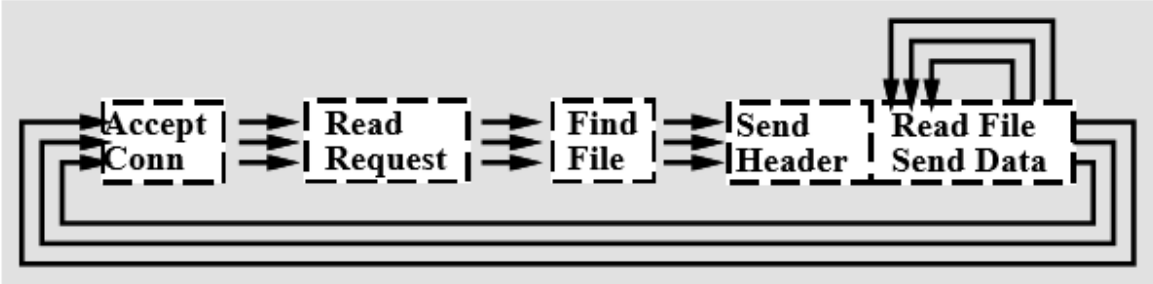


Figure 3: Multi-Threaded - The MT model uses a single address space with multiple concurrent threads of execution. Each thread handles a request.

1.3 - SINGLE-PROCESS EVENT-DRIVEN

In this paradigm, we still have just one process, but the idea here is to have as low number of threads as needed. For this, we delegate the responsibility of calling the application code to an Event Dispatcher. With this architecture, a low number of threads are capable of processing a much higher number of requests. For example, is very possible to process thousands requests with just a couple of hundreds of requests. A multiplication capacity of approximately 10 times.

Although this architecture seems to solve all process, still a problem highly affects the server performance. For example, in the figure below, we can imagine that the "Read File" box use some IO API, something like ReadFile function. Generally, these functions are synchronous, they only return when the read bytes are read to be processed. While this thread is waiting the IO, it is halted, doing nothing! Wasting computer resources. Because of this, the next paradigm was created.

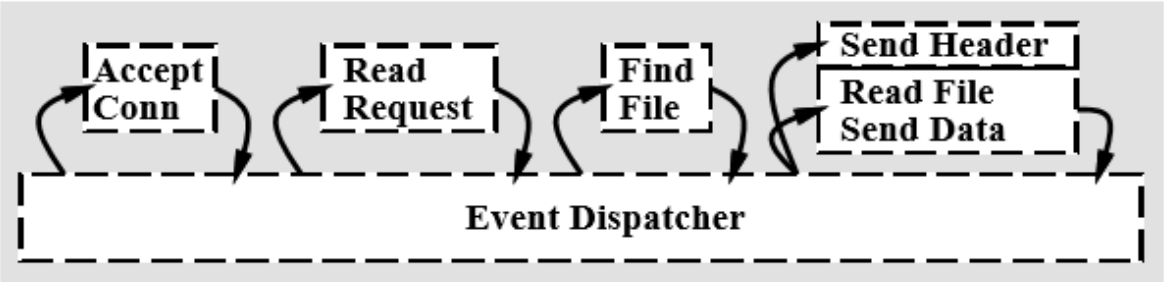


Figure 4: Single Process Event Driven - The SPED model uses a single process to perform all client processing and disk activity in an event-driven manner.

1.4 - ASYMMETRIC MULTI-PROCESS/THREAD EVENT-DRIVEN

In this paradigm, all (relevant) IO operations are made in asynchronous mode. This paradigm still delivers all the benefits of all of the others paradigms. The grand benefits that this paradigm brings to the table is that it tries to make all the available threads to work as much as needed.

So the capacity multiplier will be much higher, now it is possible to process thousands (hundreds of thousands?) of requests with as low as a couple of dozens of threads.

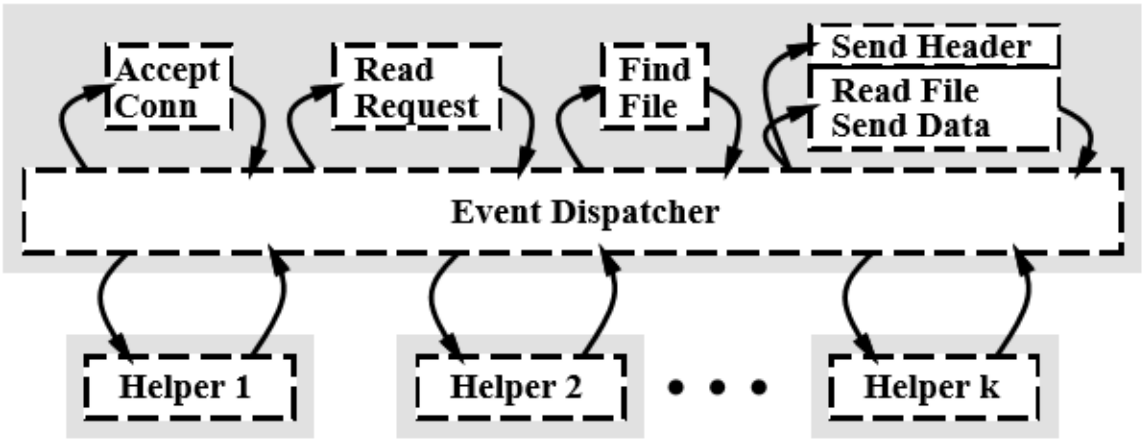


Figure 5: Asymmetric Multi-Process Event Driven - The AMPED model uses a single process for event-driven request processing, but has other helper processes to handle some disk operations.

1.5 – ANOTHER ARCHITECTURES

It is always hard to guess what the next paradigm will be. I think that the best thing that we can do is to understand all the alternatives to the current paradigm and see if the others options are better in some situations or are not mature enough.

For example, process and threads are not the only options that exists when one wants to start a flow of execution on modern OS. Windows have, for some time I must say, another option called Fibers (they also exist in the Unix world).

Fibers are very lightweight flow of executions that works on cooperative preemtiveness, on the contrary of threads that are pre-empretivivess. This means that when using fibers, one fiber flows the execution to another fibers and a fiber will never execute with any other fiber call it. A when we say lightweight we mean that the change from one fiber to other fiber can be as fast as 30ns (nanoseconds).

With this kind of performance, the problem with thread context switching trashing all CPU is gone.

For more details in Fibers se:

Fibers

https://msdn.microsoft.com/en-us/library/ms682661.aspx

Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API

https://msdn.microsoft.com/en-us/magazine/ee310108.aspx (see the 2003-September edition – it is free)

In Windows there is also another possibility: User-Mode Scheduling

On this subject, we can also take into consideration how the server will behavior when the quantity of requests are bigger than the capacity of the server. We can say that, today, we have three main strategies:

1.5.1 - DO NOTHING

Although, this is not the most technical strategy, it is the more used. The majority of networks services do not have any strategy for overload. The service will increase its load until the server get out of control.

1.5.2 - DECREASE THE SERVER QUALITY

Matt Welsh wrote a wonderful paper focusing that overload strategy is a fundamental design trait that must be architected as soon as possible on the system design

Overload Management as a Fundamental Service Design Primitive

http://www.eecs.harvard.edu/~mdw/papers/control-sigops02.pdf

1.5.3 - ELASTIC SCALABILITY

With today public/private cloud, it is much easier to elastically scale an application. That is exactly why web frontend application much be architected to work with multi-instance.

However, we must be naive to think that we can scale infinitely with this strategy. Event that the cloud infrastructure support this, cost limitation will be applied specially because in a DDOS scenario, the attack can be successful causing financial impact on the company, event on the scenario that it unsuccessfully bring your application down.

Recommended Reading

Flash: An efficient and portable Web server

https://www.usenix.org/legacy/event/usenix99/full_papers/pai/pai.pdf

A AMPED ARCHITECTURE

Given this introduction, we can analyze the complete architecture proposed by Matt Welsh, a PhD student from the Berkeley University on his thesis:

An Architecture for Highly Concurrent, Well-Conditioned Internet Services

http://www.eecs.harvard.edu/~mdw/papers/mdw-phdthesis.pdf

I consider this thesis is a seminal work on the analysis of internet services, especially because it sends us back to the discussion that we passed on the introduction. On the thesis introduction, Matt start his thesis like this:

> This dissertation presents an architecture or handling the massive concurrency and load conditioning demands of busy Internet services. Our thesis is that existing programming models and operating system structures do not adequately meet the needs of complex, dynamic Internet servers, which must support extreme concurrency (on the order of tens of thousands of client connections) and experience load spikes that are orders of magnitude greater than the average. We propose a new software framework, called the staged event-driven architecture (or SEDA), in which applications are constructed as a network of event-driven stages connected with explicit queues.

The idea of constructing distributed system using distributes components using queues are not new, off course. I can trace this idea back to other fundamental Computer Science work:

Time, Clocks and the Ordering of Events in a Distributed System

http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf

Although this work is known to have applied the concept of the inexistence of an absolute order of events, from the Special Relativity, the Leslie Lamport itself, in his site, emphasize that readers generally do not understand it. He says that one of the main points of the article was to solve this problem, to model a distributed system as a series of state machines organized with queues in front of them. So the total order of all events are simplified to the determination of the order of events on each queue.

It is explicit on the PDF page 4 on the right column (page 561 because the article was on a bigger publication). We can read the following:

> Each process maintains its own request queue which is never seen by any other process. [...] The synchronization is specified in terms of a State Machine, consisting of a set C of possible commands, a set S of possible states, and a function e: C× S -> S. The relation e(C, S) = S' means that executing the command C with the machine in state S causes the machine state to change to S'.

This idea continues on the distributed system literature with how distributed components can arrive in a consensus about the order of events, to whatever reason that they want, with Leslie Lamport proposing the Paxos Protocol.

Paxos´ paper is kind of a mystical one. A paper on consensus of distributed components started its discussion with a history about how a parliament of ancient civilization arrive in consensus.

> The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today´s fault tolerance distributed system, where legislators correspond to process and leaving the Chamber corresponds to failing.

The Part-Time Parliament

http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

This paper is considered so complicated that later Leslie Lamport wrote another paper "Paxos Made Simple" that were considered insufficient to turn Paxos an understandable protocol that another protocol was created: Raft. The paper title, "In Search of an Understandable Consensus Algorithm" clearly show the main driver of the new protocol.

Paxos Made Simple

http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf

In Search of an Understandable Consensus Algorithm (Extended Version)

http://ramcloud.stanford.edu/raft.pdf

Well... with the introduction we can start to analyze the road that ASPNET middleware has chosen for its vNext version.

Our agenda will be:

1. POSA III - Network Patterns
2. Windows API
3. libuv
4. Kestrel

CONTEXT SWITCH COST

Context Switch are expansive for various reasons:

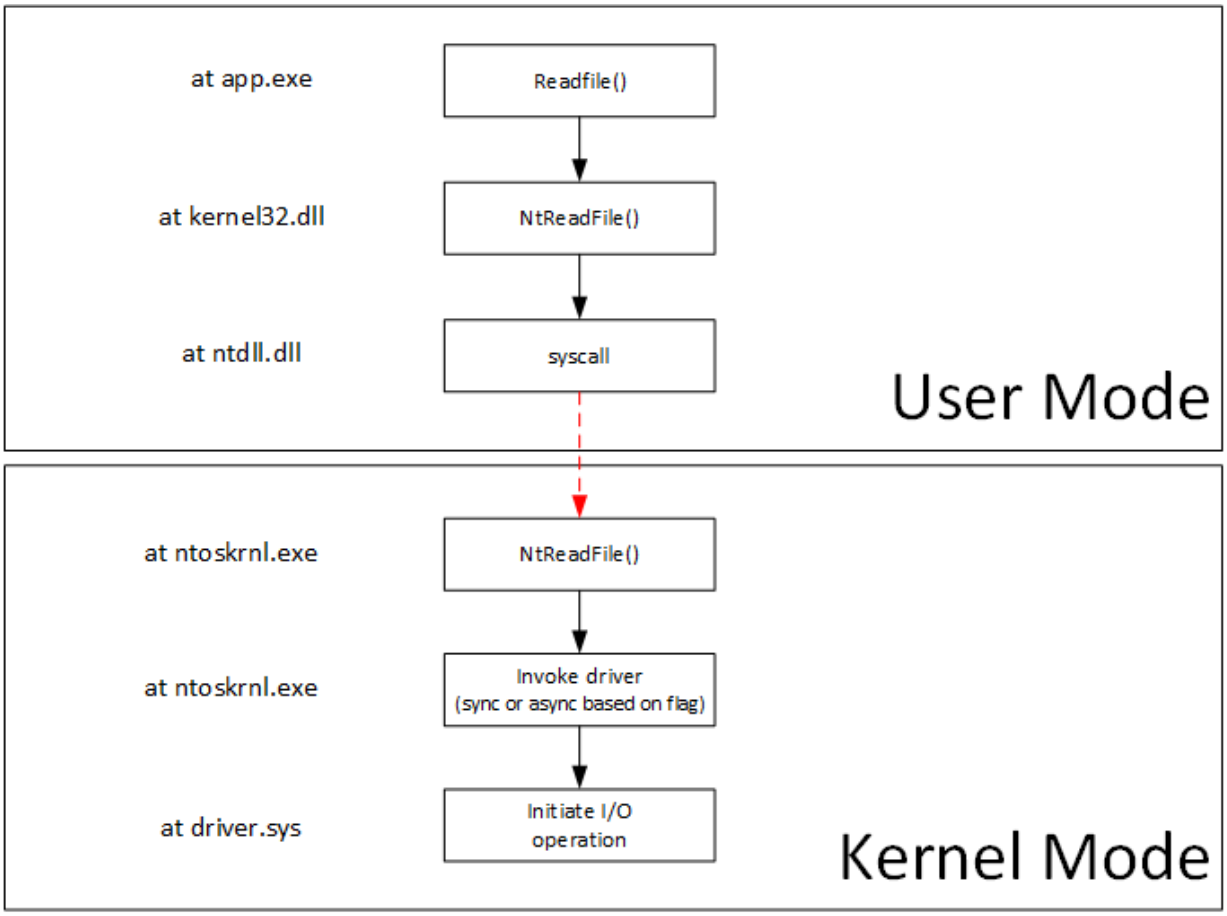
1 – syscall cost

2 – Thread/Process context switch cost

3 – Cache Trash

SYSCALL COST

The first cost associated with context switching happens when a kernel call is made. To understand this cost one must understand the OS architecture.



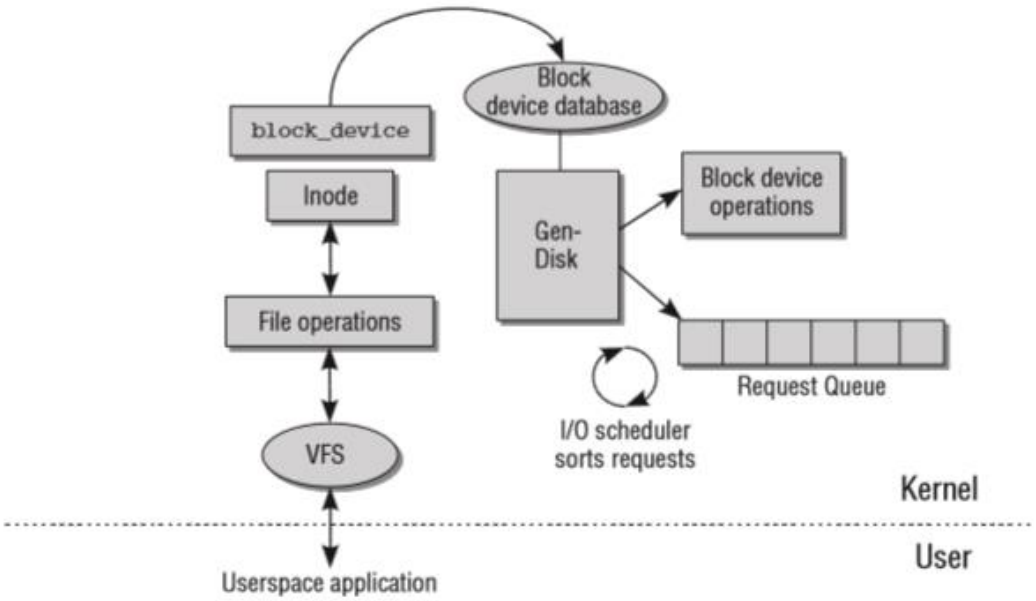


Figure 6-9: Overview of the block device layer.

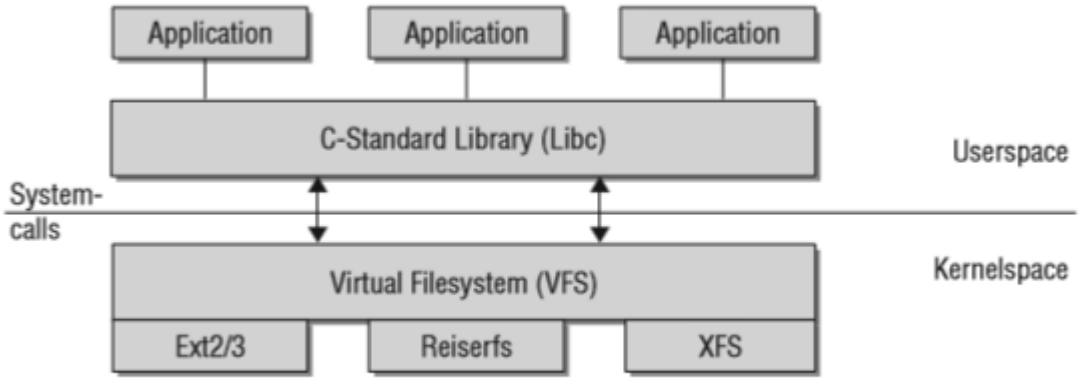


Figure 8-1: VFS layer for filesystem abstraction.

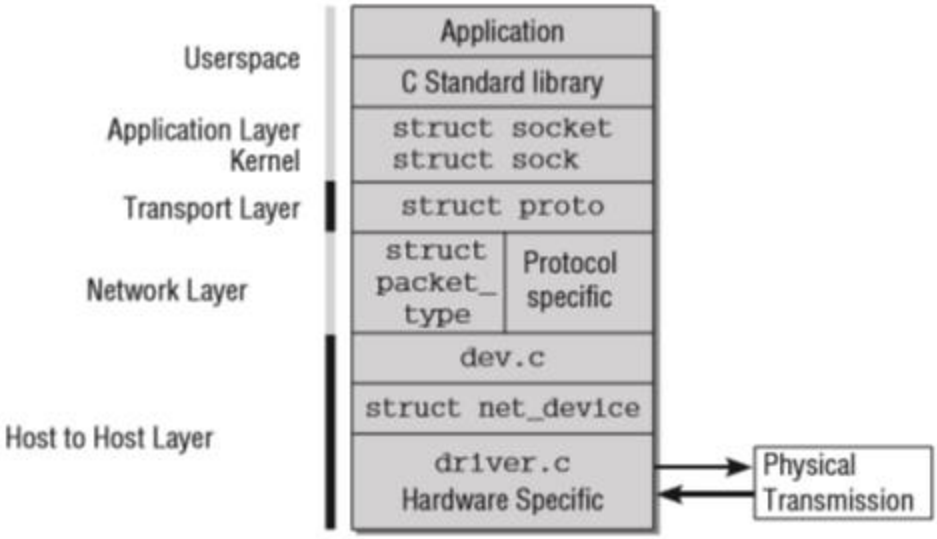


Figure 12-3: Implementation of the layer model in the kernel.

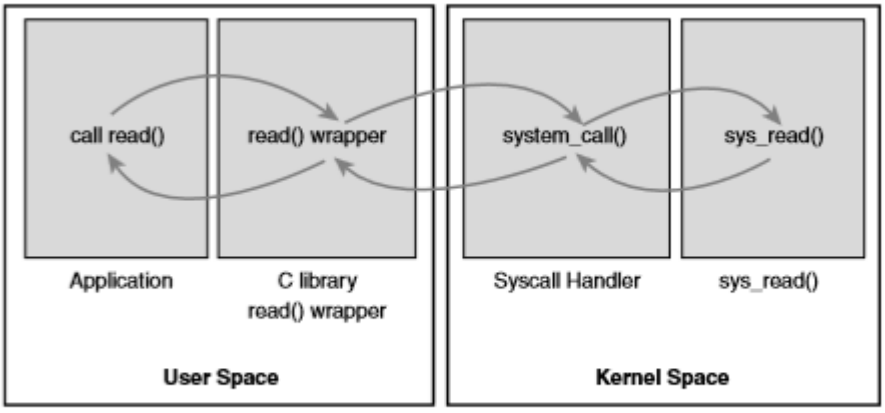


Figure 5.2 Invoking the system call handler and executing a system call.

So can see the linux kernel syscall table at:

https://github.com/torvalds/linux/blob/33caf82acf4dc420bf0f0136b886f7b27ecf90c5/arch/x86/entry/syscalls/syscall_32.tbl

```
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The abi is always "i386" for this file.
#
0       i386    restart_syscall         sys_restart_syscall
1       i386    exit                    sys_exit
2       i386    fork                    sys_fork                        sys_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open                        compat_sys_open
6       i386    close                   sys_close
```

For more details:

System Calls

http://wiki.osdev.org/System_Calls

SYSENTER

http://wiki.osdev.org/Sysenter

System Call Optimization with the SYSENTER Instruction

http://www.codeguru.com/cpp/misc/misc/system/article.php/c8223/System-Call-Optimization-with-the-SYSENTER-Instruction.htm

SYSCALL

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

Volume 2B 4- page 405

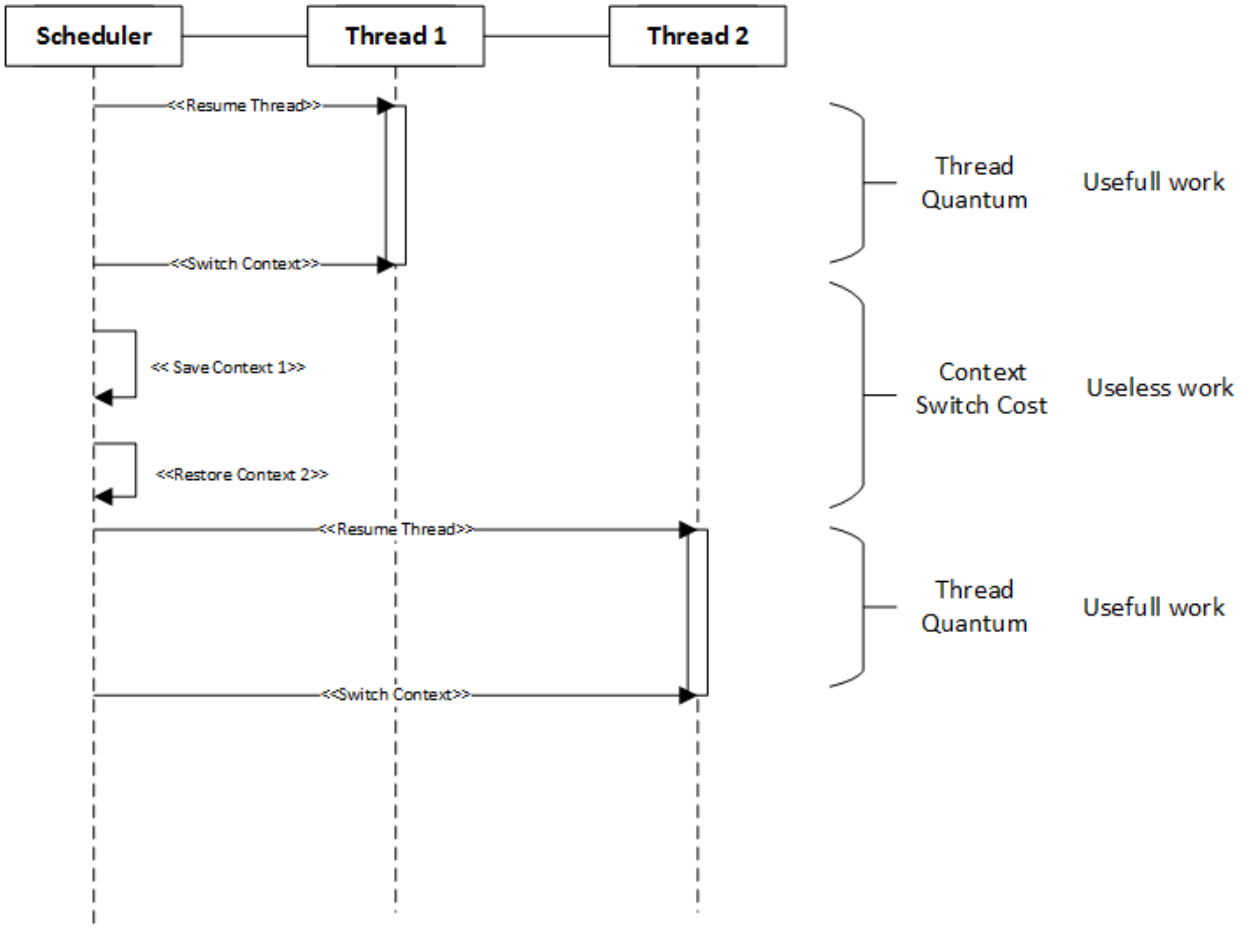
SYSENTER

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

Volume 2B 4 – page 407

THREAD/PROCESS CONTEXT SWITCH COST

http://www.cs.rochester.edu/u/cli/research/switch.pdf



On Windows one can easily monitor the quantity of Context switches easily using Performance Counters:

Monitoring Context Switches

https://technet.microsoft.com/en-us/library/cc938606.aspx

We can try to understand why a preemptive scheduler mechanism like Threading does not scale to thousands of requests when using the model of one thread per request using a theoretical queue simulation. Queue theory is a field of study from an engineer called Erlang (that is from the language name came).

http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

As we cannot access Windows source code we will sometime, access Linux source code to understand better some important concepts.

On Linux the Thread context can be seen in the file:

https://github.com/torvalds/linux/blob/d517be5fcf1a7feb06ce3d5f28055ad1ce17030b/kernel/sched/core.c#L2762

In this function that are three important points:

```
2770
2771        prepare_task_switch(rq, prev, next);
2772
```

1 - prepare_task_switch is called for each architecture that the OS supports. Most architectures do not need this functionality.

```
2786        } else
2787                switch_mm(oldmm, mm, next);
2788
```

2 - switch_mm changes the context. Depending on the processor, this is done by loading the page tables, flushing the translation lookaside buffers (partially or fully), and supplying the MMU with new information.

```
2802        /* Here we just switch the register state and the stack. */
2803        switch_to(prev, next, prev);
2804        barrier();
```

3 – switch_to is the function that actually change the execution context. As it is very architecture specific, it is almost entirely written in Assembly.

For more details:

Professional Linux Kernel Architecture

http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470343435.html

This book can be download from the Yeditepe university Operating Systems Design course (it is the BOOK #2 link):

http://cse.yeditepe.edu.tr/~kserdaroglu/spring2014/cse331/termproject/

It is impossible in a chapter talking about context switch cost and do not talk about Lazy FPU. Modern CPU architects are very aware of the cost of context switch, so they created a mode to accelerate this operation. Lazy FPU is a technique that allows the context switch algorithm to ignore all floating-point registers of being collected in the switch. This accelerates the switching.

For more details:

NetBSD Documentation: How lazy FPU context switch works

http://www.netbsd.org/docs/kernel/lazyfpu.html

This entire chapter was about the direct cost of context switching. The next chapter will talk about the indirect cost of context switching.

INDIRECT CONTEXT SWITCHING COST

TLB

[TODO]

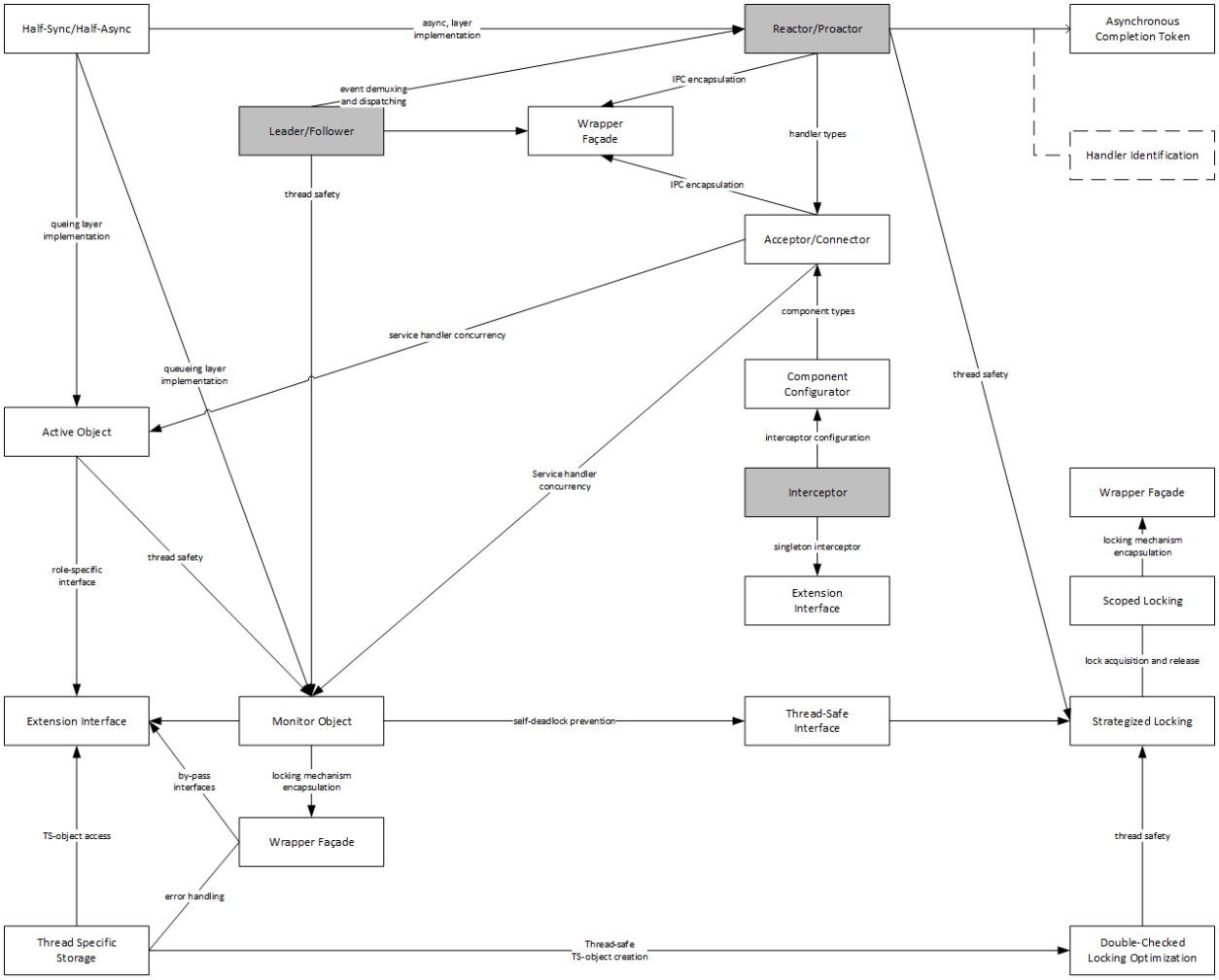
CACHE

[TODO]

1 - POSA II - NETWORK PATTERNS

POSA is a series of five books with more than 700 pages each with an incredible list of useful patterns. Unfortunately, the dynamic nature of the IT world forces the attention only to the "new kids on the block".

It is very rare to find developers or computer scientists today that have natural interest in books that are more than 10 year old. For example, my edition of the POSA I is from 1996. POSA II, the more relevant for us here, is from 2000. POSA III is from 2004. POSA IV is from 2007, the same year that POSA V was released.



This fact is very sad to me, because the simply reading about 50 pages a lot of developers would achieve a much higher level of maturity when analyzing server architecture.

First, let us start with the Reactor pattern on POSA II, page 154.

The Reactor patterns is an Object-Oriented design to architect the "Event Dispatcher" of the figure above of the AMPED architecture. Off course, on other programming paradigms exists others solutions more appropriated. C#, the main language of .NET is a multi-paradigm language and the Reactor Pattern can be design differently. Nevertheless, we must start somewhere and the POSA design is a wonderful start.

POSA describe the Reactor as:

> The Reactor architectural pattern (179) allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients. The structure introduced by the Reactor pattern 'inverts' the flow of control within an application, which is known as the Hollywood Principle.

For Hollywood Principle see:

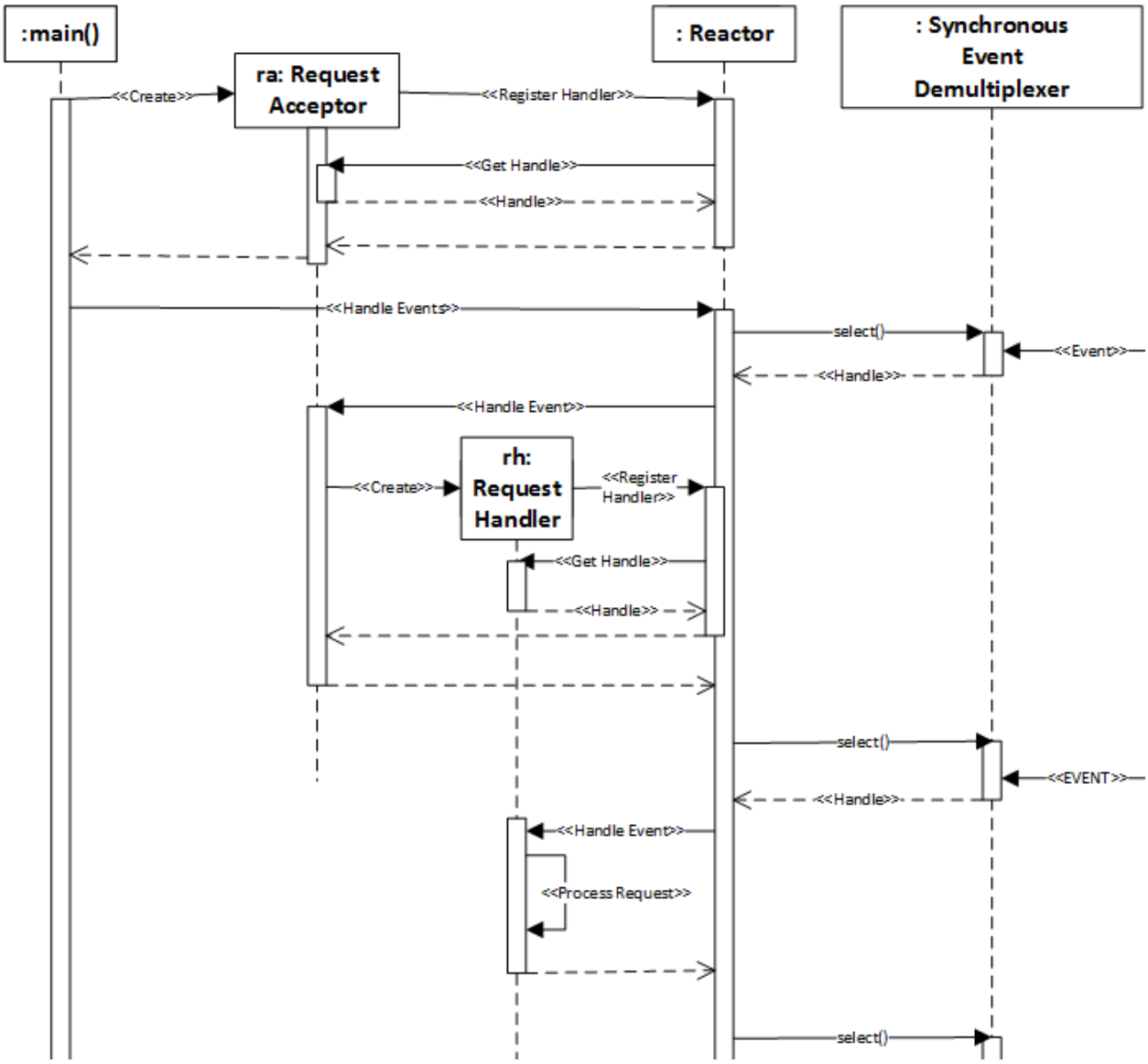
Pattern Hatching: Design Patterns Applied

http://www.amazon.com/Pattern-Hatching-Design-Patterns-Applied/dp/0201432935

For many, this principle is what defines the difference between frameworks and libraries. So we can that the Reactor is a pattern for writing concurrent event-driven frameworks.

Below there is a simplified UML Sequence Diagram of how a Reactor works. Using the Reactor patterns generates the following consequences:

- Benefits
    - Separation of Concerns
    - Modularity
    - Reusability
    - Configurability
    - Portability
    - Concurrency Control
- Liabilities
    - Restricted Applicability
    - Non Pre-emptiviness
    - Complexity



In the above diagram, the Synchronous Event Demultiplexer is the Windows API. In the following chapters, when we talk in more details about the Windows API we will see the “select” and other Windows API. Demultiplexing in this context means the capacity of waiting for various events (http request, file IO etc…) with just one call.

A wonderful implementation of the Reactor pattern can be found on the ACE framework and its internals can be studied in this paper

The Design and Use of the ACE Reactor: An Object-Oriented Framework for Event Demultiplexing

http://www.cs.wustl.edu/~schmidt/PDF/reactor-rules.pdf

The Proactor is another event-based pattern. It is very similar to the Reactor and the POSA example of a server using the Proactor is a web server, serving static html files. Very appropriate, I would say.

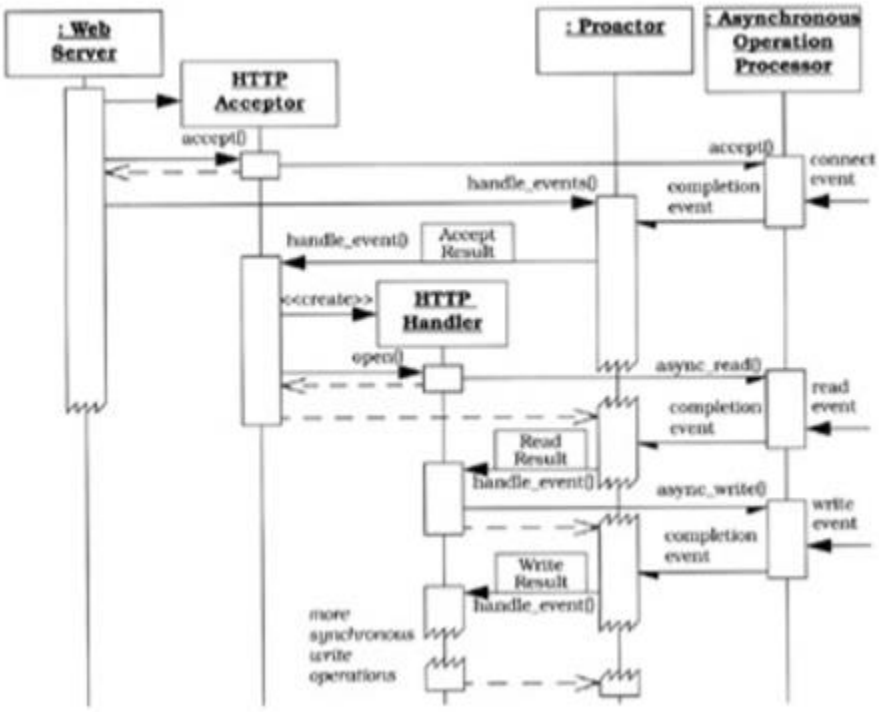
The Proactor architectural pattern allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations. It offers the performance benefits of concurrency without incurring some of its liabilities

So, one can see that it is almost the same of the Reactor pattern. The main differences of them are that

> The Proactor supports the demultiplexing and dispatching of multiple event handlers that are triggered by the completion of asynchronous operations. In contrast, the Reactor pattern is responsible for demultiplexing and dispatching multiple event handlers that are triggered when indication events signal that it is possible to initiate an operation synchronously without blocking.

The Proactor example, the webserver, is mainly using Windows APIs because at the time some UNIX distributions did not have any asynchronous APIS. Off course, this is not the case anymore. Both, Windows and Unix APIs have evolved and I hope to show how the Windows APIs have evolved.

Here is the UML Sequence Diagram of how a Web Server could be implemented using the Proactor pattern.



Using the Proactor One have the following consequences:

- Benefits
  - Separation of Concerns
  - Portability
  - Encapsulation of Concurrency mechanisms

        - Decoupling of Threading and Concurrency
        - Performance
        - Simplification of Synchronization
    - Liabilities
        - Restricted Applicability
        - Complexity of Programming and Debugging
        - Scheduling, Controlling and Canceling Asynchronously Running Operations

For more information on the Proactor pattern, one can see:

Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events

http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf

2 - WINDOWS API

As I said in the Proactor part, the POSA book used the Windows API as example because the async IO API on Windows has been in production for a very long time.

For this, we will use the book

Windows Internals, Part 2

http://www.amazon.com/Windows-Internals-Edition-Developer-Reference/dp/0735665877

WINDOS API HISTORY

[TODO]

JOBS, PROCESS, THREADS, AND FIBERS

[TODO]

IO API

In Windows there is no such thing as synchronous IO [quote Windows Internals].

Mark Russinovich suggest this flow of control to illustrate when a read operation is initiated (an observation is very important: to Windows every IO operation is asynchronous. When you in your application ask for a synchronous operation, Windows block your thread only).

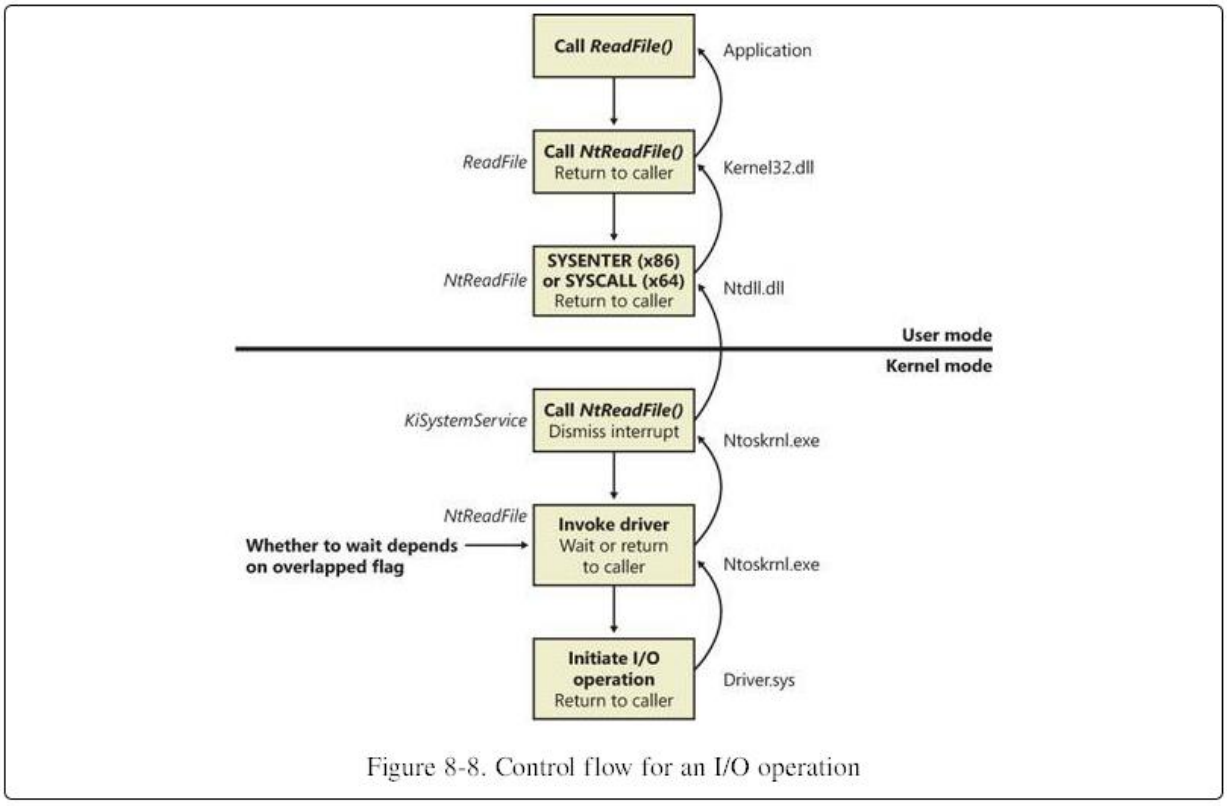


Figure 8-8. Control flow for an I/O operation

NETWORK API

The following diagram is a map of all Windows components regarding network organized using the OSI Model. For this article, just some of these boxes are of our interest: Winsock 2.0 API, WinHTTP API, IIS, HTTP.sys and Winsock kernel.

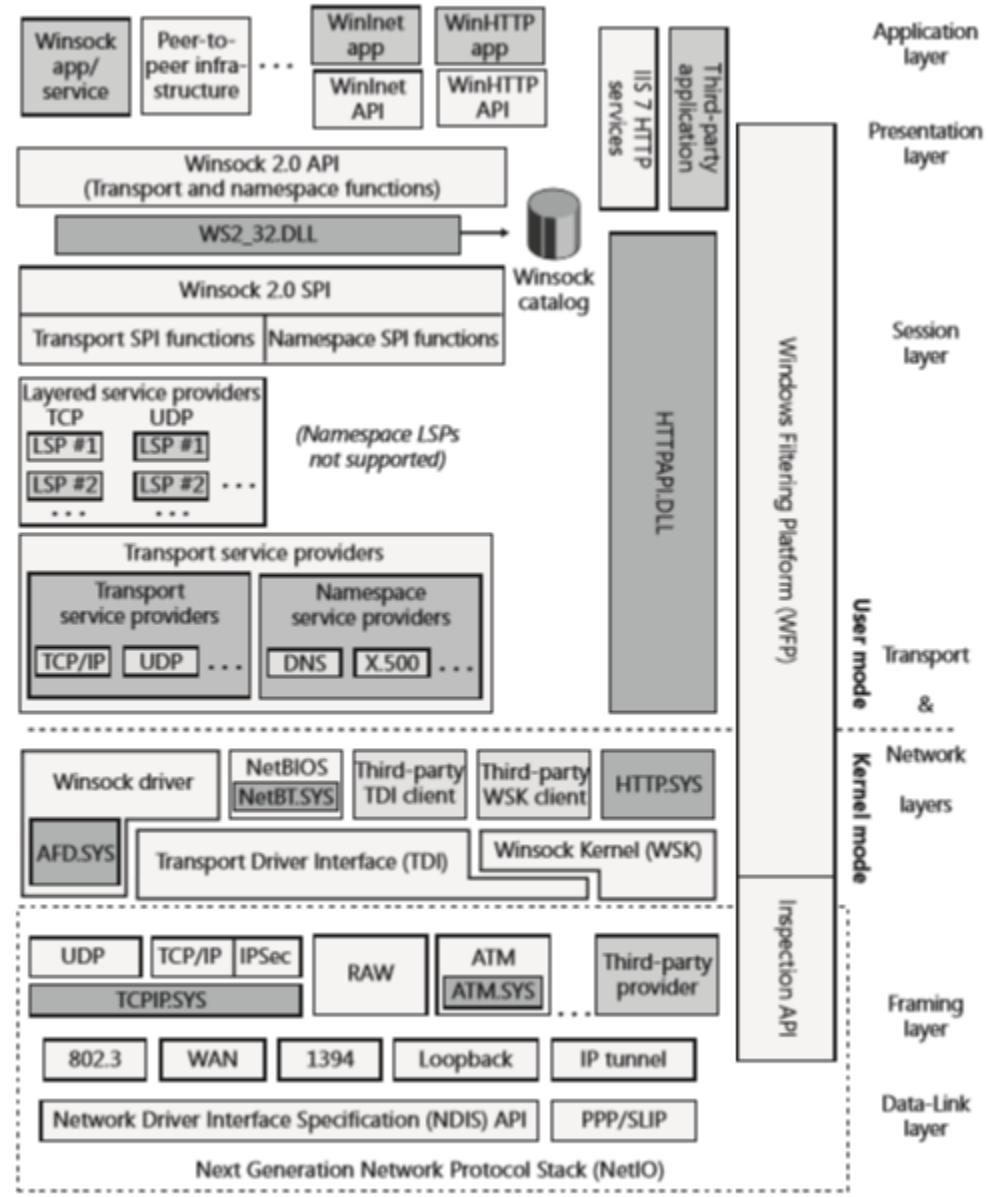


FIGURE 12-2 OSI model and Windows networking components

WinSock is the Microsoft implementation of the BSD Sockets API. The original specification is from the 80´s, so is not a surprise that WinSock implements many enhancements over the original API. Today, WinSock is at version 2.2. For the same reason that exists WinSock, Microsoft implemented WinSock Kernel, a network API that is used in Kernel Mode for drivers.

Another important component is the WinHTTP API. The current version 6.0 is a Http Server implementation talks directly to the Http.Sys driver. One of its main features are to support Port Sharing and cache without exiting Kernel Mode.

The last component is IIS, the default web server that Microsoft offer. Although, ASPNET vNext will not depends on IIS, given that Microsoft want that ASPNET be portable, the default architecture in Windows do include IIS and its module Http Platform Module.

All low-level network API in Windows are asynchronous, even when called throughout sync/blocking calls. This exists for two reasons:

1 – Sync/blocking is easier to program against;

2 – In some situations is can be faster.

This pattern in known as Half-Sync/Half-Async. It is another famous pattern from the POSA series. It first appeared in the POSA II book and again in the POSA IV book. For more details please read:

Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-structured Concurrent I/O

http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf

Half Sync/Half Async

http://www.cs.wustl.edu/~schmidt/PDF/HS-HA.pdf

COMPLETION PORTS API

One can create a Completion Port in Windows using the method CreateIoCompletionPort
https://msdn.microsoft.com/en-us/library/windows/desktop/aa363862(v=vs.85).aspx

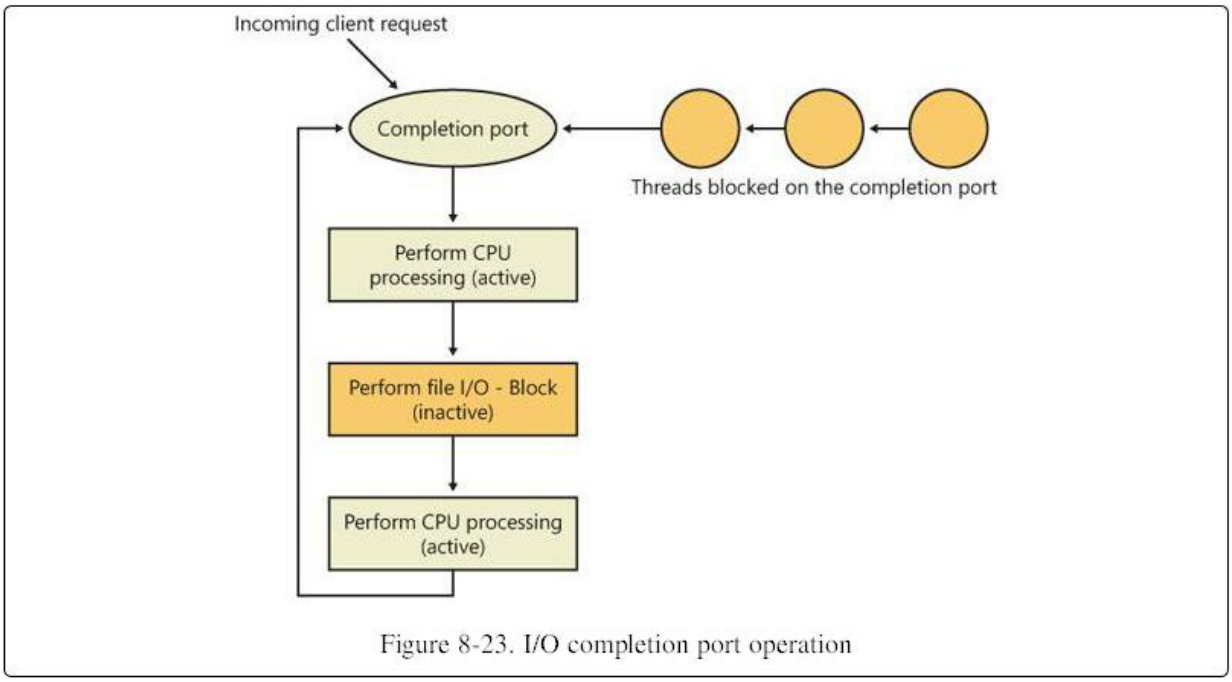


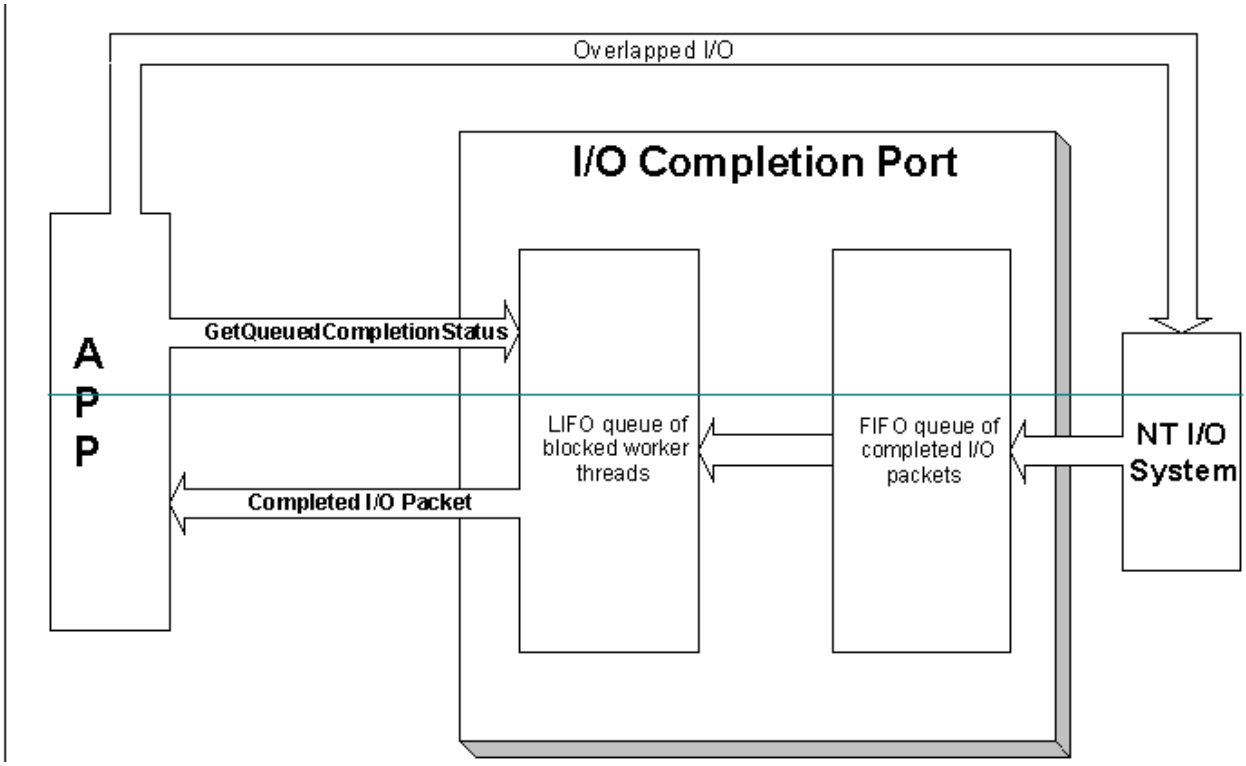
Figure 8-23. I/O completion port operation

To be honest, I do not think that one can understand the model using IO Completion ports through this diagram, but it has an important detail. An attentions reader can read this diagram and realize that the threads queue is all blocked waiting for the completion port to have any IO packet. You may be thinking: “Have I read this entire article, being convinced that blocking threads are bad so that you show me a model with blocked threads?”

Yeah! Kind of difficult answer.

The idea of the architectures shown here are not to not have blocked threads, but to minimize the quantity of blocked threads. For example, those three blocked threads on the diagram are very capable, on a busy server, to generate work for hundreds, even thousands of requests.

Moving on, I think that this diagram show on a super high level the idea of completion ports.



For a tutorial on how to use I/O Completion port, please see:

A simple application using I/O Completion Ports and WinSock

http://www.codeproject.com/Articles/13382/A-simple-application-using-I-O-Completion-Ports-an

Well, I/O Completion Ports were the best mode to build busy servers in Windows and have existed by a long time. They were event used in the POSA II book.

However, the Windows Kernel version 6.1 brought a new API that promises even better scalability for Servers (Yeah, sometimes it is useful to update the Windows…)

REGISTERED I/O

Although it is in production code from almost 4 years now, it is very rare to listen anything about Registered IO. There are two sites to understand what they are.

Registered Input/output (RIO) API Extensions

https://technet.microsoft.com/en-us/library/hh997032.aspx

New techniques to develop low-latency network apps

https://channel9.msdn.com/events/Build/BUILD2011/SAC-593T

The idea behind Registered I/O is that there is a huge waste of processing power just moving memory from one location (kernel space) to another (user space). This idea has already appeared, for example, in this paper.

Fbufs: A High-Bandwidth Cross-Domain Transfer Facility

http://zoo.cs.yale.edu/classes/cs422/2010/bib/druschel93fbufs.pdf

TCP LOOPBACK OTIMIZATION

There are many IPC mechanisms in Windows. For sure, the fastest in using Shared Memory, but using TCP Loopback is for sure one of the most portable solutions. One can easily separate the sender and the receiver to another machine and the communications would still work.

For this reason, Windows Kernel 6.1 implemented an optimization that will increase the usefulness of TCP loopback in IPC with low latency needs.

For more information about please see:

Fast TCP Loopback Performance and Low Latency with Windows Server 2012 TCP Loopback Fast Path

http://blogs.technet.com/b/wincat/archive/2012/12/05/fast-tcp-loopback-performance-and-low-latency-with-windows-server-2012-tcp-loopback-fast-path.aspx

3 – LIBUV

[TODO]

4 – KESTREL AND ASPNET

[TODO]