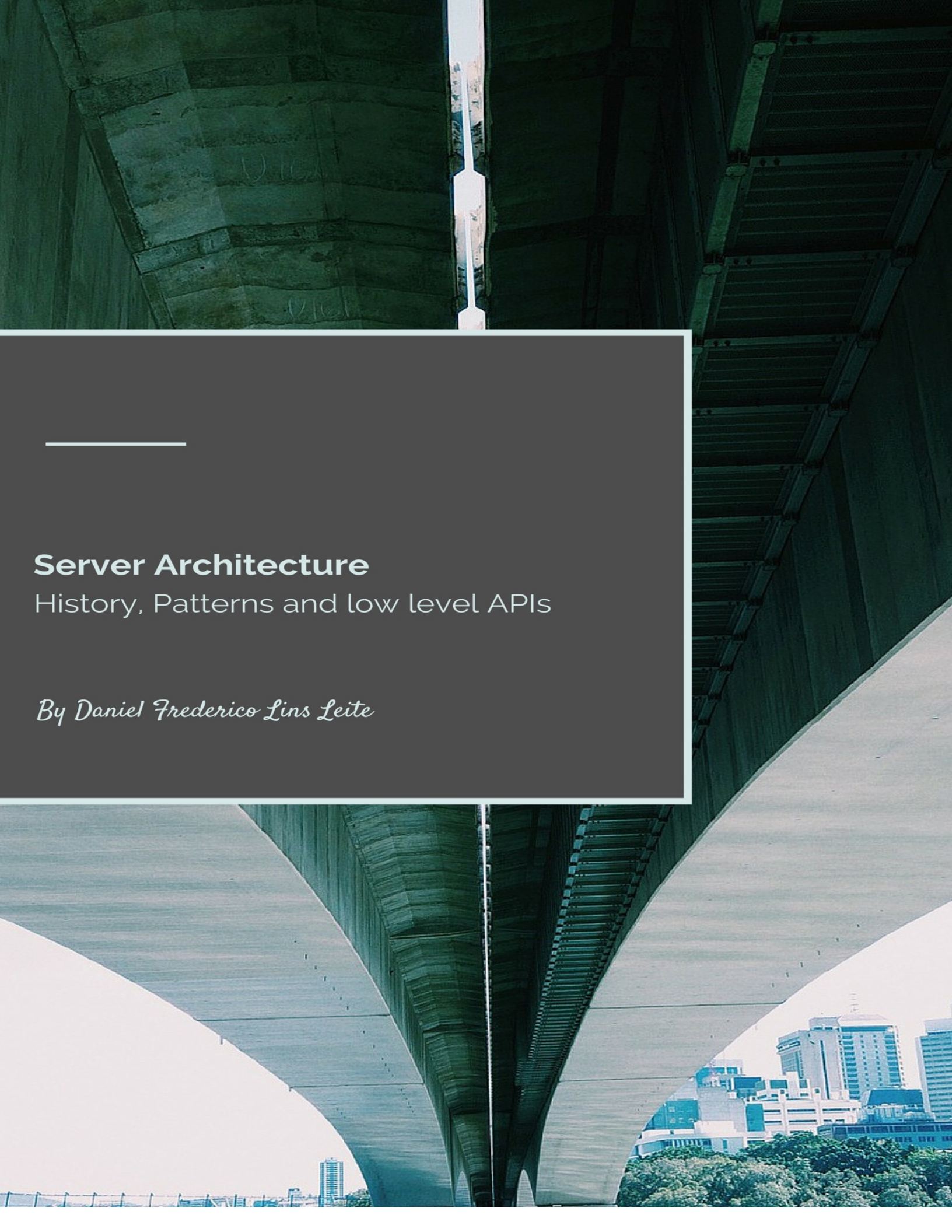


Server Architecture

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....5
Server Middleware Architecture Evolution....7
1.1 Multi Process Paradigm....7
1.2 - Multi Thread Paradigm....8
1.3 - Single-Process Event-Driven....9
1.4 - Asymmetric Multi-Process/Thread Event-Driven....10
1.5 – Another Architectures....11
A AMPED Architecture....13
Context Switch Cost....15
syscall Cost....15
Thread/Process Context Switch Cost....19
Indirect Context Switching Cost....21
TLB....21
Cache....21
1 - POSA II - Network Patterns....43
2 - Windows API....48
Windos API History....48
Jobs, Process, Threads, and Fibers....48
IO API....48
Network API....49
Completion Ports API....50
Registered I/O....52
TCP Loopback Otimization....52
Scheduler architecture....53
Simples implementation....53
One queue, many processors....53
Many QUEUES, MANY PROCESSORS....53
Synchronized Queue....53
Two queues: public sync, private Non-Sync....53
Thread-safe push queue (MPSC)....53
Working-stealing queues....54

3 – Algorithms......55
4 – Testing......56

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.

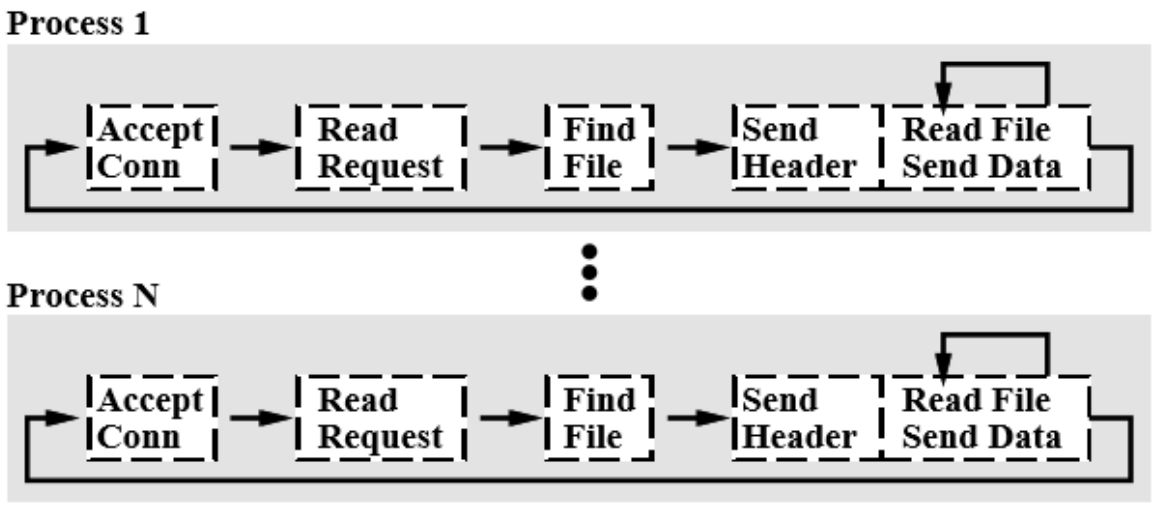


Figure 2: Multi-Process - In the MP model, each server process handles one request at a time. Processes execute the processing stages sequentially.

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.

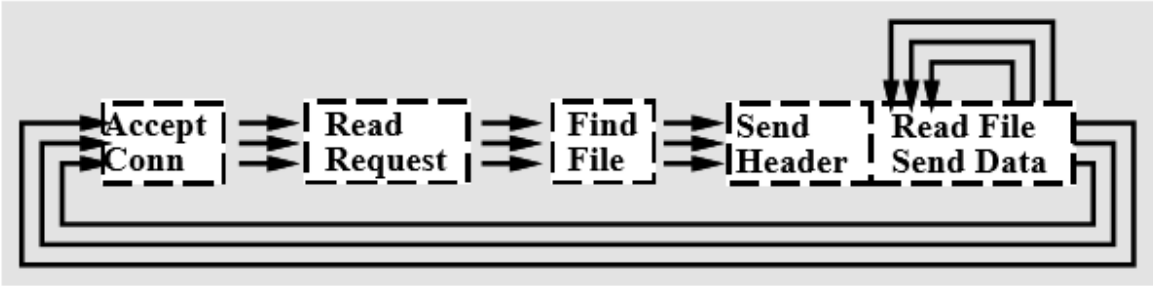


Figure 3: Multi-Threaded - The MT model uses a single address space with multiple concurrent threads of execution. Each thread handles a request.

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.

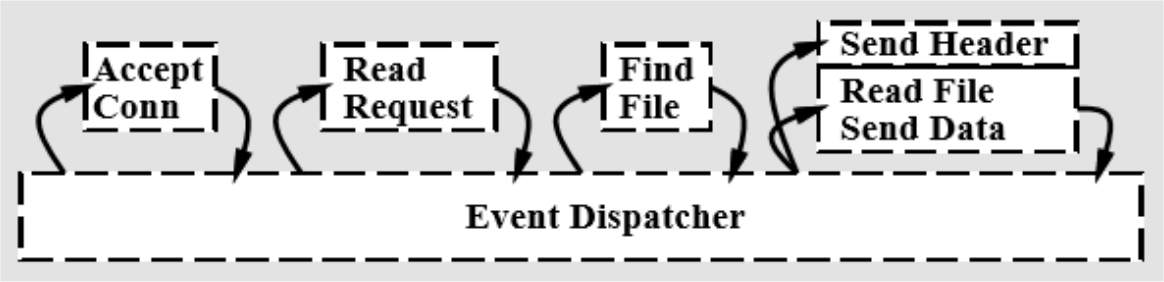


Figure 4: Single Process Event Driven - The SPED model uses a single process to perform all client processing and disk activity in an event-driven manner.

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.

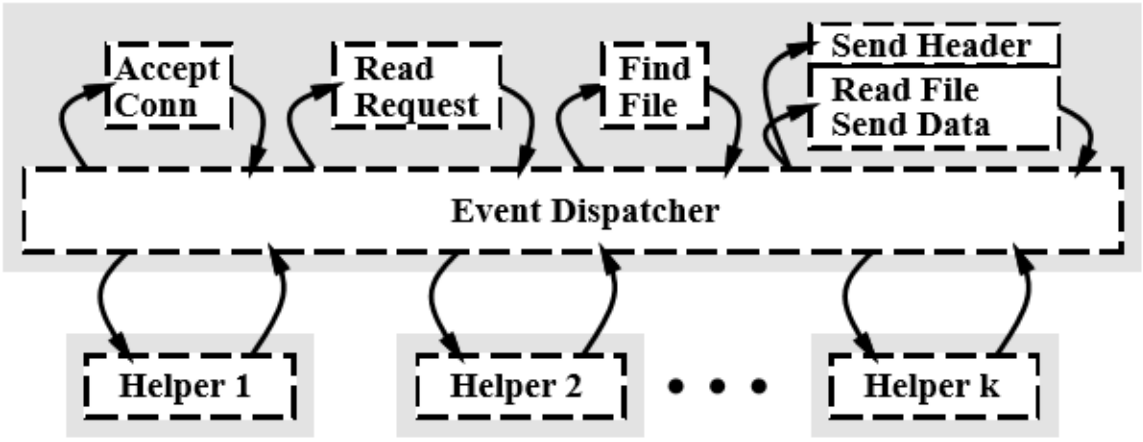


Figure 5: Asymmetric Multi-Process Event Driven - The AMPED model uses a single process for event-driven request processing, but has other helper processes to handle some disk operations.

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:

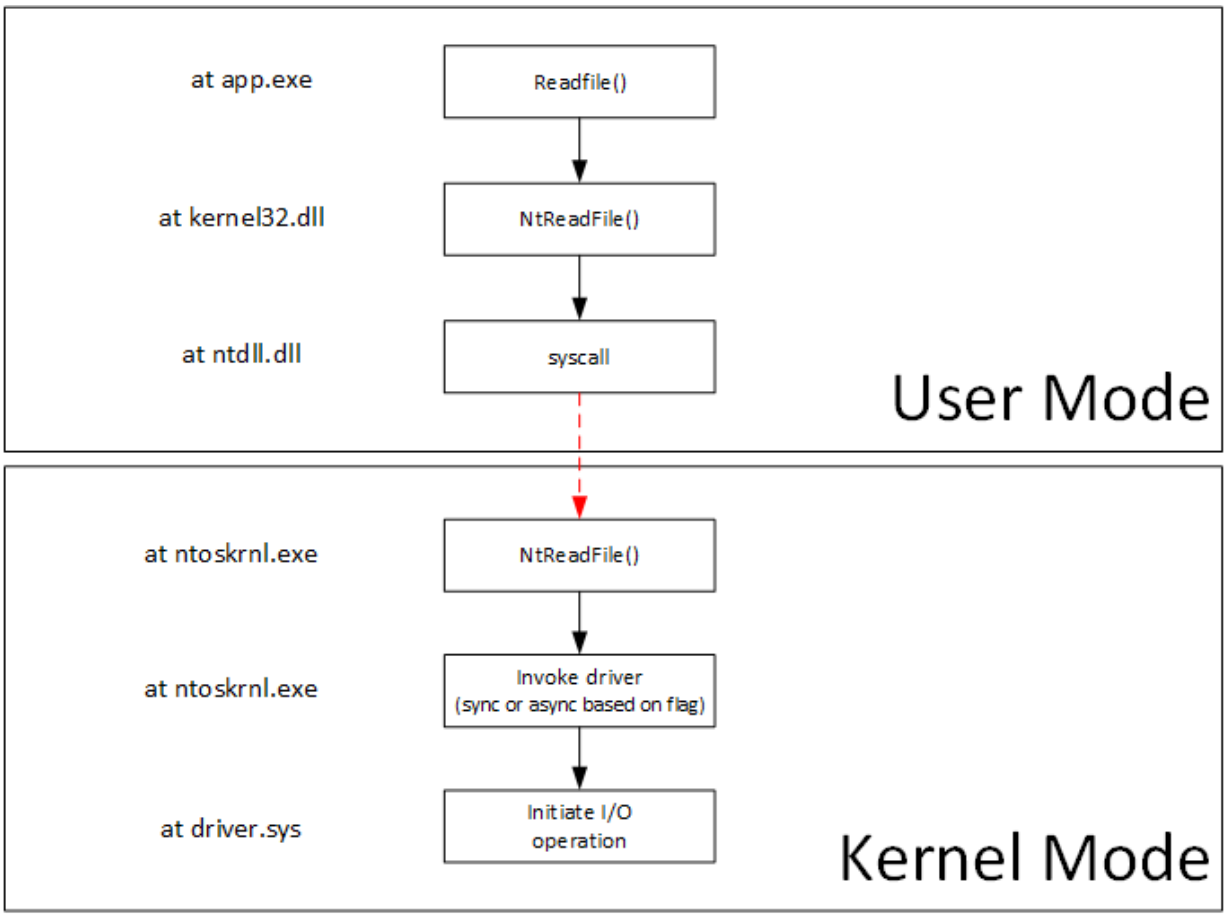
1 – syscall cost

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.



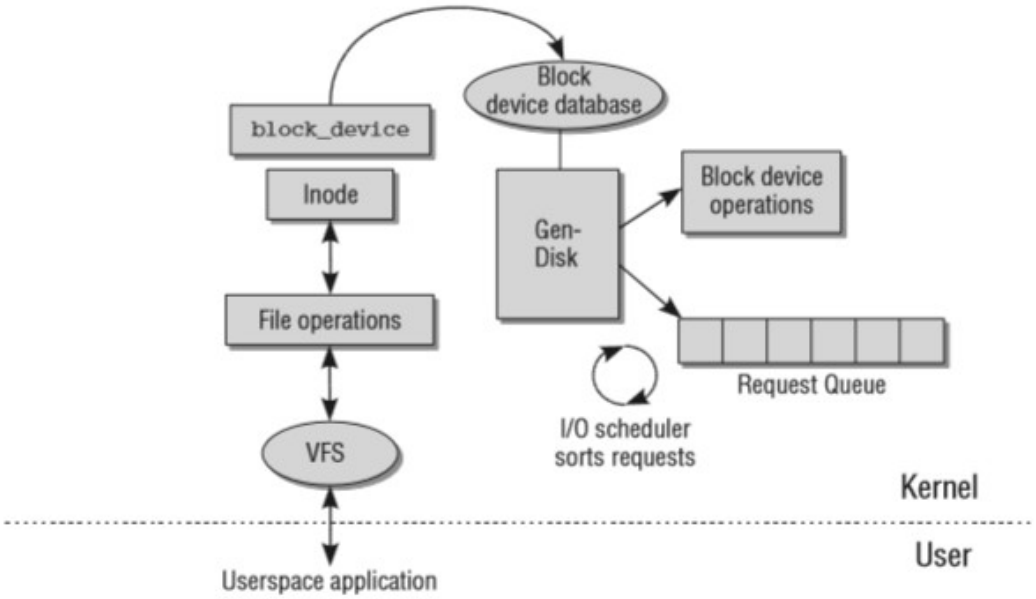


Figure 6-9: Overview of the block device layer.

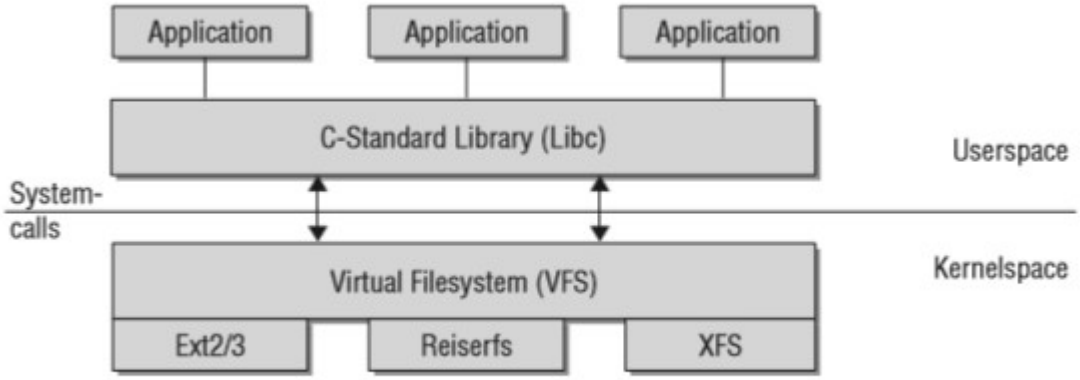


Figure 8-1: VFS layer for filesystem abstraction.

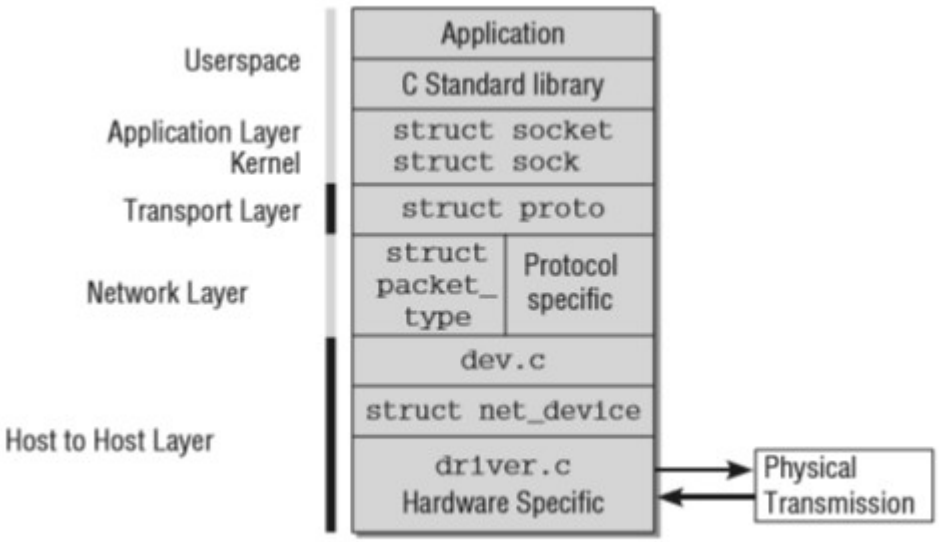


Figure 12-3: Implementation of the layer model in the kernel.

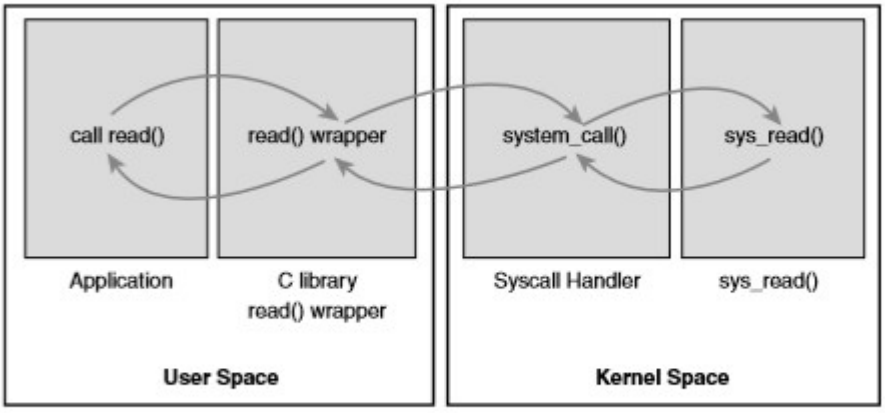


Figure 5.2 Invoking the system call handler and executing a system call.

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

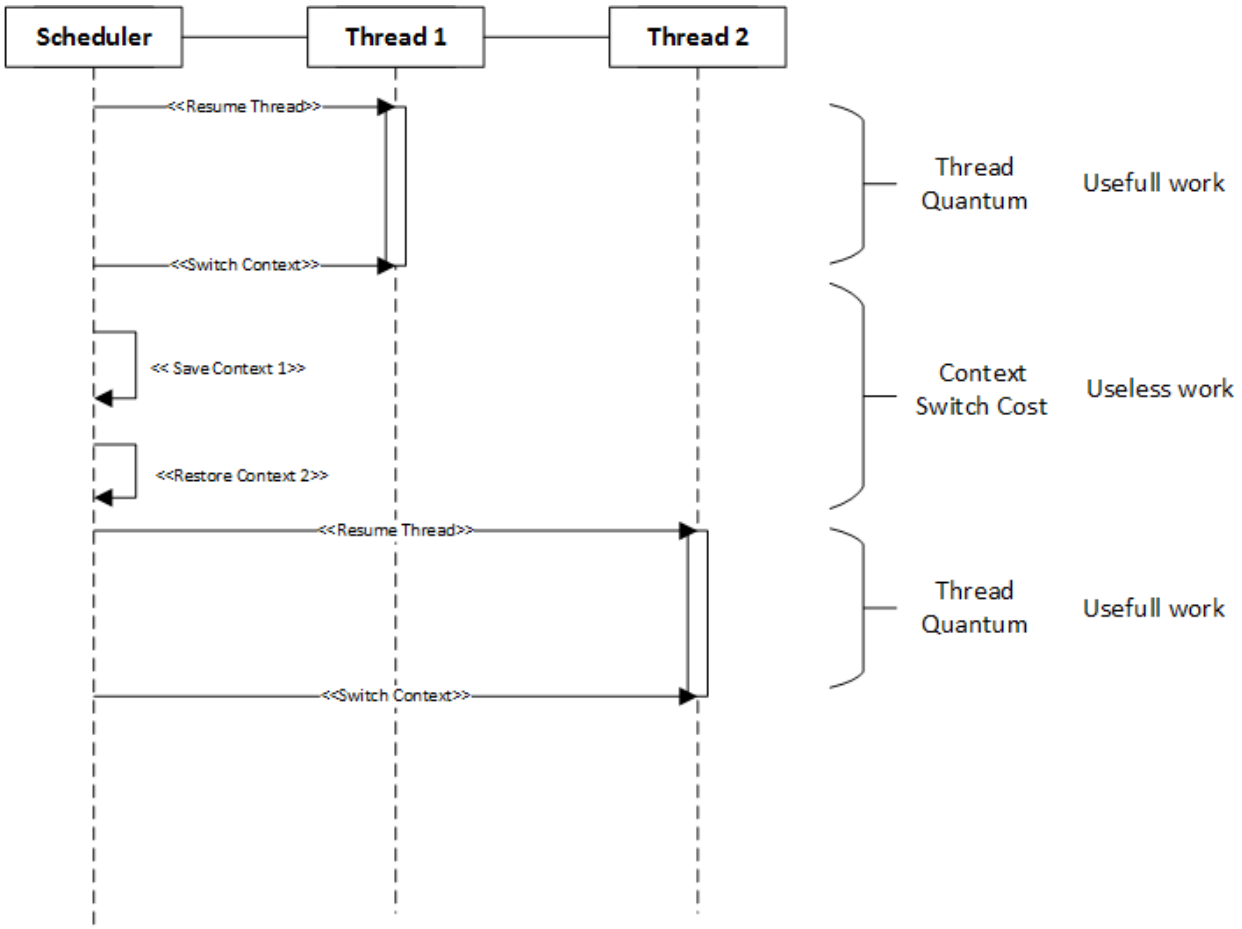
SYSENTER

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

To understand the impact that cache has on execution, let us change to a more synthetic, simpler example. We are going to analyze the impact of cache misses when summing a float value that is:

1 – packed in float array;

2 – inside the property "x" in a struct array.

Our little program is something like this:

```
...
struct SomeData
{
   char junk[32];
   float x;
   char doesntMatter[32];
};
..
int main(int argc, char** argv)
{
   ...
        float* f = new float[size];

        float accum = 0;
        for(int i = 0; i < size; ++i)
        {
           accum += f[i];
        }

        delete[] f;
   ...
        SomeData* arr = new SomeData[size];

        float accum = 0;
        for(int i = 0; i < size; ++i)
        {
           accum += arr[i].x;
        }

        delete[] arr;
   ...
}
```

```

Our little program accepts two parameters: "col" and/or "row". If we pass "col", it runs the "sum" on the float array.

```
float* f = new float[size];

float accum = 0;
for(int i = 0; i < size; ++i)
{
   accum += f[i];
}

delete[] f;
```

If we pass "row" it runs on the struct.

```
SomeData* arr = new SomeData[size];

float accum = 0;
for(int i = 0; i < size; ++i)
{
   accum += arr[i].x;
}

delete[] arr;
```

The program will output a "CSV" table with the columns:

1 - The arguments. One in each line;

2 - A csv table with three columns;

2.1 - Size of the array;

2.2 - Time in seconds to sum the "col" (float array);

2.3 - Time in seconds to sum the "row" (struct array).
```

So, for example.

```
> ./a.out col row
0:./a.out
1:col
2:row
size;colunar;row
1000;0;0;
10000;0;0;
100000;0;0.006;
1000000;0.006;0.065;
10000000;0.068;0.684;
```

Maybe this is a surprise for you, but the "col" case (float array) is 10 times faster! It took 0.684 seconds again 0.068. If you are surprised, and want to know what is happening here, please continue reading.

VALGRIND

To gather information to understand what is happening here, we will use "Valgrind".

Valgrind is a virtual-machine that record information about memory read, write and cache (and other information). Maybe it is not clear why we gathering this information to understand this case. It will become clear at the end, I hope.

But one of the reasons why we are investigating memory is stated inside the "Valgrind" manual.

> On a modern machine, an L1 miss will typically cost around 10 cycles,
>
> an LL miss can cost as much as 200 cycles, and a mispredicted branch
>
> costs in the region of 10 to 30 cycles. Detailed cache and branch
>
> profiling can be very useful for understanding how your program
>
> interacts with the machine and thus how to make it faster.
>
> https://valgrind.org/docs/manual/cg-manual.html

If you do not remember, or do not know, what a cache miss is, let us make a quick recap first.

ASM, CPU ARCHITECTURE RECAP

The way that CPUs work is through instructions. Each command is store in binary, off course, but for human readability they have mnemonics. For example:

```
ADD RAX, RBX
```

https://www.felixcloutier.com/x86/add

This instruction would just do "RAX += RBX". Where "RAX" and "RBX" are register inside the CPU. You can imagine CPU register as variables (in this case ints) that reside inside the CPU and so have "zero" cost of access.

Although it is insanely difficult to actually estimate the cost of an instruction, a "ADD" like this using only registers takes just one cycle.

```
Instruction Operands    Ops Latency
ADD, SUB    r,r/i       1   1
page 28
```

https://www.agner.org/optimize/instruction_tables.pdf

So a computer with an 1GHz cpu would do 1 billion of "ADD"s like this one. The problem is, how do you put the values we want to sum inside the CPU registers. To do this you have to use another instruction:

```
MOV RAX, [MEM ADDR]
MOV RBX, [MEM ADDR]
```

https://www.felixcloutier.com/x86/mov

So with these we move from some memory location to those registers. After this we can sum them. Let us ignore for now how do I know the address that I want to load from.

```
MOV RAX, [0x100]
MOV RBX, [0x104]
ADD RAX, RBX
```

Moving cost a little more than summing two registers.

```
Instruction Operands    Ops Latency
MOV         r64,m64     1   3
page 28
```

https://www.agner.org/optimize/instruction_tables.pdf

"Latency" three means that it takes, as the "Agner" documentations states, at minimum three cycles to complete. In this particular case is the very-very-minimum, because as we saw in "Valgrind" manual, it can actually take hundreds of cycles. If you are really unlucky it can actually take thousands and thousand of cycles (some milliseconds, but let us ignore this for now).

Tallying up how many cycles out little program would take, we have:

```
MOV RAX, [0x100]    # 200 cycles
MOV RBX, [0x104]    # 200 cycles
ADD RAX, RBX        # 1 cycle
```

401 cycles. But...

Because accessing memory is to slow from the CPU perspective, CPU architects envisioned faster layers of memory to speed this access up. So instead of always going to the biggest/slowest memory, the CPU caches and tries to access the same information in faster memories. Today is common to have three levels of memory.

These levels generate a pyramid of memory. At the very top are CPU registers, very fast, but very limited. Only 15 for integers; the second level is called L1. Is a very fast memory. Slower than registers, but almost as good. But they also small.

For example my computer has 32KB for L1 cache. Better than 15 ints, but insanely small for today standards. What program could you make using only 32KB?

```
> getconf -a|grep LEVEL1_DCACHE
LEVEL1_DCACHE_SIZE             32768
LEVEL1_DCACHE_ASSOC            8
LEVEL1_DCACHE_LINESIZE         64
```

Below L1 is, off course, L2. Slower to access, but bigger. In my case 512KB. And L3, 16MB. Already interesting.

```
> getconf -a|grep LEVEL2
LEVEL2_CACHE_SIZE              524288
LEVEL2_CACHE_ASSOC             8
LEVEL2_CACHE_LINESIZE          64
> getconf -a|grep LEVEL3
LEVEL3_CACHE_SIZE              16777216
LEVEL3_CACHE_ASSOC             16
LEVEL3_CACHE_LINESIZE          64
```

The main difference here, is that L1 exist inside each CPU chip; L2 is outside but exist for each CPU; and the L3 is shared across all CPUs. This makes a huge difference.

So to get more real, more physical, let us dive deep into an AMD CPU. Only because is the CPU that I am using it now.

```
> cat  /proc/cpuinfo|grep name
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
```

```
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
model name      : AMD Ryzen 7 1700 Eight-Core Processor
```

You can read all at: https://en.wikichip.org/wiki/amd/microarchitectures/zen

But for a more succinct description, and what matters to us here you can just continue.

INSIDE THE CCX (CORE COMPLEX)

Here we have a "photo" of chip, a part of the CPU.

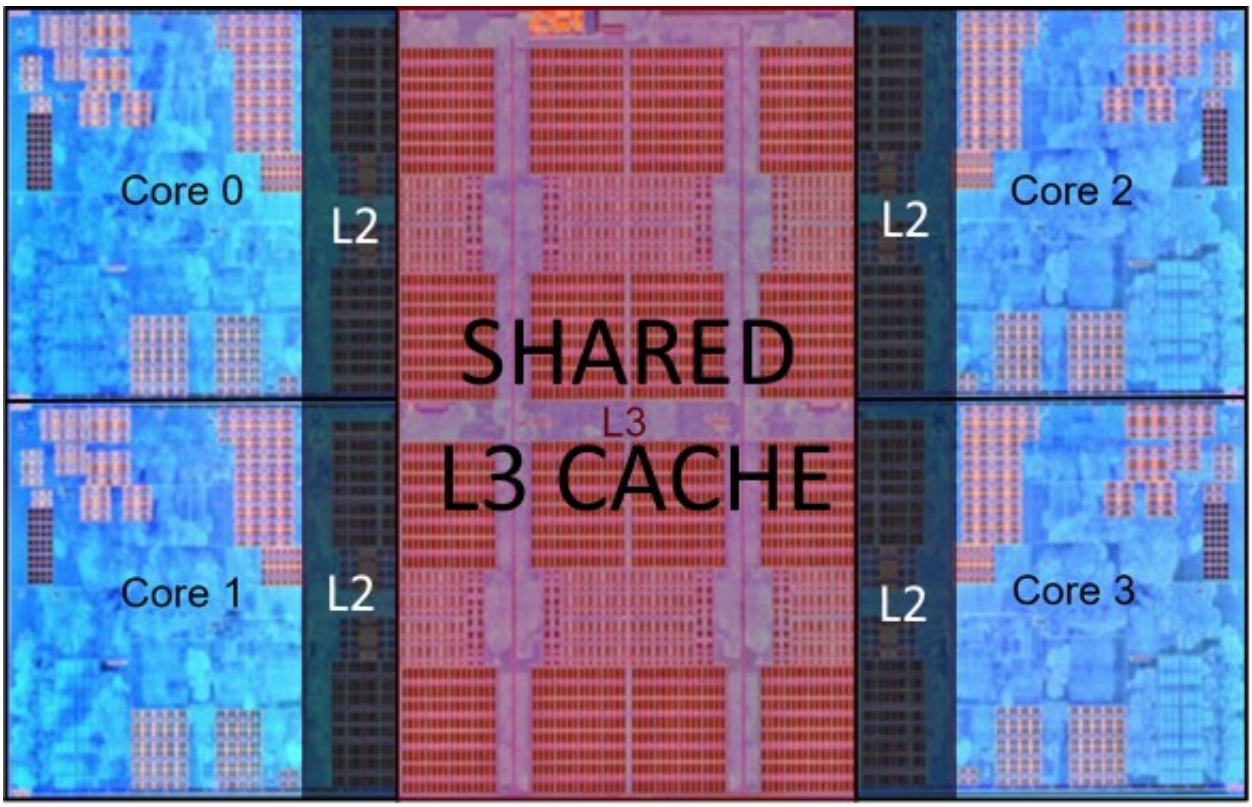
1 - We are seeing four cores;

2 - Cache L3 is shared, and takes a lot of the CCX space;

3 - Cache L2 is not inside the core but is replicated to each core;

4 - We are not seeing cache L1 because it sits inside the core.

Given that my computer has 8 physical cores, I have two of these. How they communicate is not important here.



INSIDE THE CORE

The things to understand here are:

1 - Cache L2 outside, but near the core. 512KB for each core;

1.1 - "L2CTL" means "Level 2 Controller". It decides if the memory location you a reading is inside the L2 (using the L2$Tags) and in which "bank" its located (L2$Bank1 or L2$Bank 2);

2 - "L1I$" is the L1 cache ONLY for instructions;

3 - "L1D$" is the L1 cache ONLY for data;

4 - ALU is the chip the runs Arithmetic instructions with integers;

5 - FPU is the chip the runs Arithmetic instructions with floats;

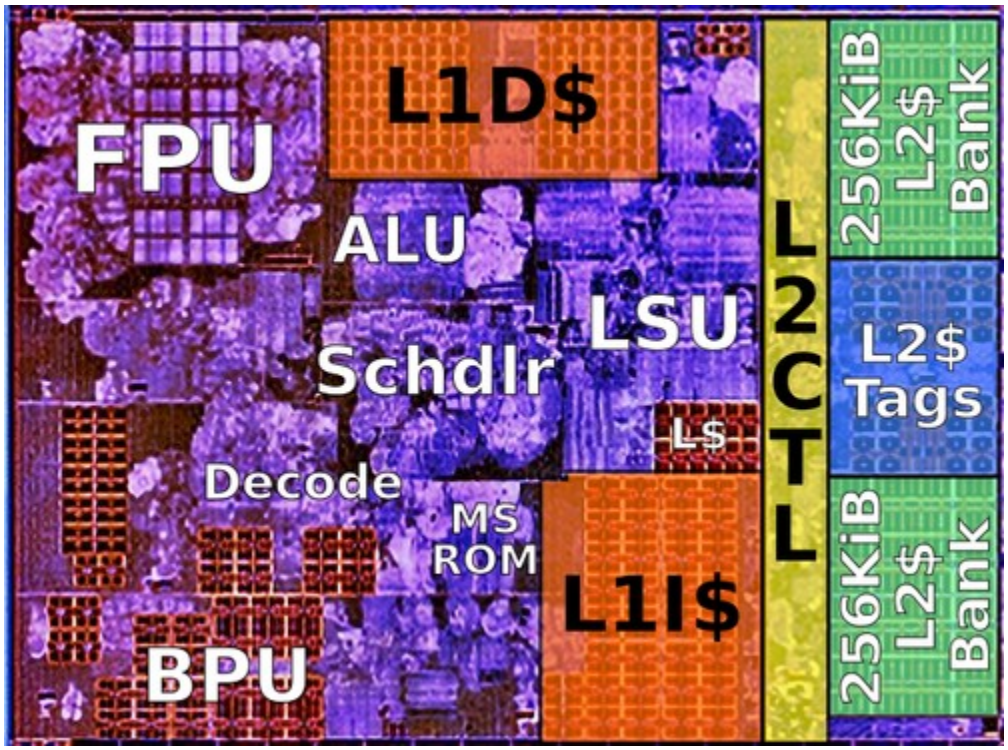
6 - BPU is the chip that optimize decisions (if, for, function calls , jumps etc...);

7 - LSU is the chip that manage load/store from memory.

7.1 - L$ is the store of what is inside the L1 cache;

8 - "Decode" is the chip that decode the binary instructions to actual CPU commands;

9 - "Schdlr" means scheduler and is the chip that organize everything.



It is clear that things that work together are near together.

- ALU and FPU work with registers (inside its own chips) and memory, so they are near the "Data Cache";

- BPU and Decode work with instructions so they are near the "Instruction Cache";

- When we have a "Level 1 Cache Miss" we pass the read to the "Level 2". So both "Instruction Cache" and "Data Cache" are near the LSU

It is also clear why higher levels of cache are slower. It not only a matter of "better" or "faster" memory. They are behind increasing number of chips.

If you are "summing" two memories inside the "Data Cache" you request passed through:

1 - LSU

2 - LSU verifies if the memory address is inside the L$;

3 - Read from "L1D$"

If you need to go to L2 you have:

1 - LSU;

2 - LSU verifies if the memory address is inside the L$;

3 - L1 Cache Miss;

4 - L2CTL;

5 - LSCTL verifies if the memory address is inside the "L2$ Tags";

6 - Read from "L2$ Bank 1" or "L2$ Bank 2".

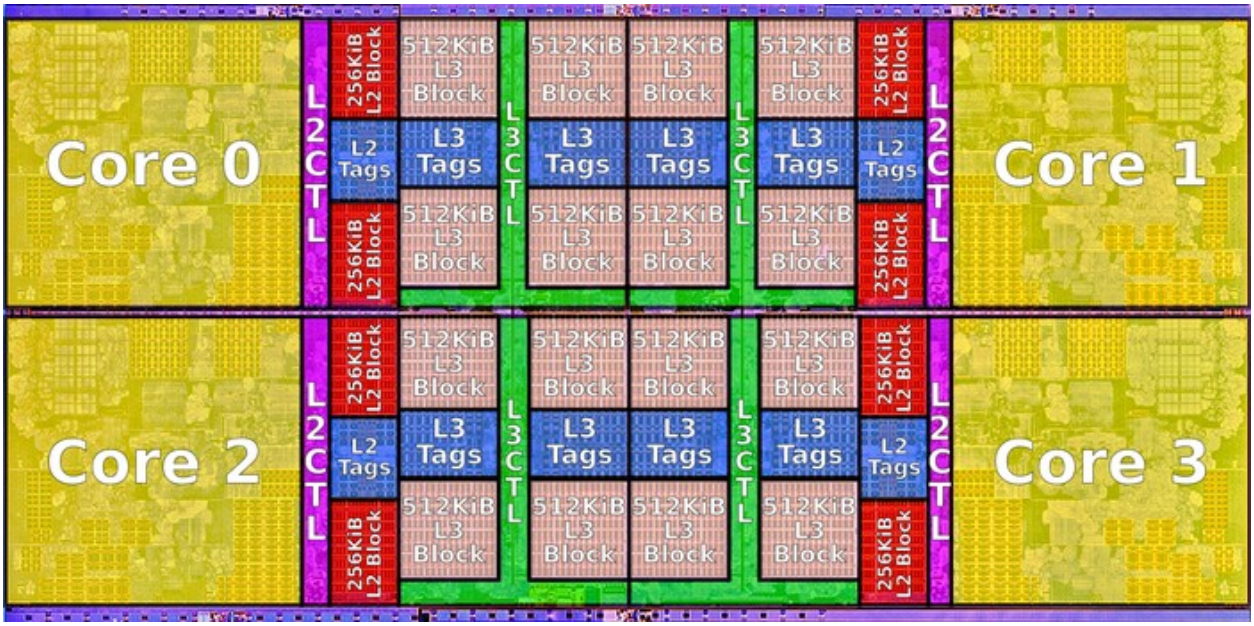
and the same thing happen when we go to Level 3. The difference here is that "Level 3" is shared. Which means we can have contention.

Let us zoom on the CCX again. If the step 5 fail and we have a "L2 Cache Miss" we need to go to the "L3 Cache" or as "Valgrind called it" "Last Level Cache".

Level 3 is shared, in my case here between four cores. The Same logic applies here. "Level 2 Controller" pass the read to "Level 3 Controller.

But suppose that by pure chance all four "Level 2 Controllers" pass at the exact same time reads to the "Level 3 Controller" to read the same memory address. They will all go for the same "L3 Block" and we will have read contention.

Off course we do not need to read the exact same memory address to have contention here. We just need to have the bad luck to end in the same "L3 Block".



Enough of theory, let us see the program running.

EXECUTION

Now let us see how the "col" execution.

VALGRIND ABOUT "COL" EXECUTION

```
> valgrind --tool=cachegrind ./a.out col
==3987== Cachegrind, a cache and branch-prediction profiler
==3987== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==3987== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3987== Command: ./a.out col
==3987==
--3987-- warning: L3 cache found, using its data for the LL simulation.
==3987== error calling PR_SET_PTRACER, vgdb might block
0:./a.out
1:col
size;colunar;row
1000;0;
10000;0;
100000;0.005;
1000000;0.055;
10000000;0.546;
==3987==
==3987== I   refs:      135,639,615
==3987== I1  misses:          1,941
==3987== LLi misses:          1,871
==3987== I1  miss rate:        0.00%
==3987== LLi miss rate:        0.00%
==3987==
==3987== D   refs:      100,773,214  (78,342,402 rd   + 22,430,812 wr)
==3987== D1  misses:        711,908  (   709,360 rd   +      2,548 wr)
==3987== LLd misses:        641,864  (   640,335 rd   +      1,529 wr)
==3987== D1  miss rate:         0.7% (       0.9%     +        0.0%  )
==3987== LLd miss rate:         0.6% (       0.8%     +        0.0%  )
==3987==
==3987== LL refs:           713,849  (   711,301 rd   +      2,548 wr)
==3987== LL misses:         643,735  (   642,206 rd   +      1,529 wr)
==3987== LL miss rate:          0.3% (       0.3%     +        0.0%  )
```

Let us break this output line-by-line, when necessary.

```
> valgrind --tool=cachegrind ./a.out col
```

This is our command line, off course. "./a.out" is our executable name. "col" means that we are running just the "col" or packed example.

```
==3987== Cachegrind, a cache and branch-prediction profiler
==3987== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==3987== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3987== Command: ./a.out col
```

Here "Valgrind" is just confirming our command line. So far so good.

```
--3987-- warning: L3 cache found, using its data for the LL simulation.
```

Ok, here we have the first important info. "Valgrind" identified that my computer have three levels of cache. These days, L1 and L2 are inside the core chip, therefore are faster.

L3, level three, is shared between all cores.

In a NUMA environment the L3 is not shared across NUMAs, just between cores of one of the NUMAs.

The important part here is that L3 is being considered as LL, the "Last Level" of cache. We will see some statistics about this later.

```
==3987== error calling PR_SET_PTRACER, vgdb might block
```

We will ignore this for now.

```
0:./a.out
1:col
size;colunar;row
1000;0;
10000;0;
100000;0.005;
1000000;0.055;
10000000;0.546;
```

This is the output of our program.

First two lines are the arguments passes. Confirming that we are running just the "col" for now. Then we have the timing. Everything as expected. Every time we increase the size by 10 we take 10 times more time. It is good when things make sense.

```
==3987==
==3987== I   refs:     135,639,615
==3987== I1  misses:         1,941
==3987== LLi misses:         1,871
==3987== I1  miss rate:       0.00%
==3987== LLi miss rate:       0.00%
```

"I" here is for the instruction cache. I1 is for the level one cache for instruction, and LLi is for the "last level" of cache for instructions. In our case here we now that this is L3.

"I refs" so is the number of instructions read. Off course that our program does not have 135M of instructions. We achieve this number because of our loops. The interesting thing about this number is that, sometimes, less instructions means faster code. If you decrease the number of instructions without adding branches, indirections etc... you are probably optimizing your code. OK, so we read 135M instructions.

"I1 misses" is of course how many times we tried to find the "next instruction" on the L1 for instruction and failed. The number itself if hard to analyze if it is good or bad. Again when comparing implementations decreasing this number is good.

"LLi misses" is really THE problem when comes to instruction cache miss, because we failed to find our code on the "last cache".

"I1 miss rate" and "LLi miss rate" are probably the important ones here, because they summarize both misses in terms of how many instructions were read. In out case here, "0.00%" is an approximation because we had cache misses, off course, but they are negligible. In summary they are just ("I1 misses"/"I refs" and "LLi misses"/"I refs").

```
==3987== D   refs:     100,773,214  (78,342,402 rd   + 22,430,812 wr)
==3987== D1  misses:       711,908  (   709,360 rd   +     2,548 wr)
==3987== LLd misses:       641,864  (   640,335 rd   +     1,529 wr)
==3987== D1  miss rate:        0.7% (      0.9%     +       0.0%  )
==3987== LLd miss rate:        0.6% (      0.8%     +       0.0%  )
```

"D" is about the data memory cache.

"D   refs" is how many times the memory was touched. "rd" how many times was read, and "wr" how many times was written. In out case 100M times, being 78M reads and 22M writes.

"D1 misses" is how many times we missed our data on the L1 cache. Again aggregated (711K times) and by reads (709K) and writes (2k) times. Up until now it is pretty clear that we are "read dominated". Even if you find a magical way to decrease the number of writes, it would not make a huge difference.

"LLd misses" is how many times we failed on the last level of cache. In terms of memory time cost, this is the big villain. Every time you magically decrease this number you will see improvements in performance.

"D1 miss rate" and "LLd miss rate" are, off course, in comparison with the amount of memory read/writes. We are pretty good actually. And this is the whole point of this article. When we access data in a contiguous array, as the "cache predictor" is expecting, we have very little L1 and L3 misses and this improves our performance A LOT. This, in a lot of cases, beats algorithmic complexity.

```
==3987== LL refs:          713,849  (   711,301 rd   +     2,548 wr)
==3987== LL misses:        643,735  (   642,206 rd   +     1,529 wr)
==3987== LL miss rate:         0.3% (       0.3%     +       0.0%  )
```

The last section is just a summary.

"LL refs" is the sum of "LLi refs" and "LLd refs" (both do not appear in the output).

"LL misses" is the sum of "LLi misses" and "LLd misses".

"LL miss rate" is a misleading one. It is NOT "LL misses"/"LL refs" but "LL misses"/("I refs"+"D refs").

Now let us see how the "row" stands against "Valgrind".

VALGRIND ABOUT "ROW" EXECUTION

```
> valgrind --tool=cachegrind ./a.out col
> valgrind --tool=cachegrind ./a.out row
==4053== Cachegrind, a cache and branch-prediction profiler
==4053== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==4053== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4053== Command: ./a.out row
==4053==
--4053-- warning: L3 cache found, using its data for the LL simulation.
==4053== error calling PR_SET_PTRACER, vgdb might block
0:./a.out
1:row
size;colunar;row
1000;0.001;
10000;0.001;
100000;0.013;
1000000;0.129;
10000000;1.371;
==4053==
==4053== I   refs:      157,863,039
==4053== I1  misses:          1,940
==4053== LLi misses:          1,880
==4053== I1  miss rate:        0.00%
==4053== LLi miss rate:        0.00%
==4053==
==4053== D   refs:      100,773,688  (78,342,700 rd   + 22,430,988 wr)
==4053== D1  misses:     11,128,717  (11,126,135 rd   +      2,582 wr)
==4053== LLd misses:     11,021,134  (11,019,573 rd   +      1,561 wr)
==4053== D1  miss rate:        11.0% (      14.2%     +       0.0%  )
==4053== LLd miss rate:        10.9% (      14.1%     +       0.0%  )
==4053==
==4053== LL refs:        11,130,657  (11,128,075 rd   +      2,582 wr)
==4053== LL misses:      11,023,014  (11,021,453 rd   +      1,561 wr)
==4053== LL miss rate:          4.3% (       4.7%     +       0.0%  )
```

This time we will not analyze the output line-by-line, off course. We will go directly to the values and compare them. First let us start with timing. We saw that "col" was much faster, let us see if "Valgrind" preserve this.

```
size;colunar;row
1000;0;
10000;0;
100000;0.005;
1000000;0.055;
10000000;0.546;
```

versus

```
1000;0.001;
10000;0.001;
100000;0.013;
1000000;0.129;
10000000;1.371;
```

And yes, it is much worse. The absolute value here does not matter, because we are running a debug compilation using "Valgrind". But it is easy to see that "col" is much better. And this IS the reason of this article and let us see why now.

First let us see if the difference lies on number of instructions:

```
==3987== I   refs:      135,639,615
```

versus

```
==4053== I   refs:      157,863,039
```

Well, the "row" version is indeed worst in this dimension. But it is enough to explain almost 3x worse? Probably not. Let us jump to the others statistics and can see later the difference in the generated asm.

```
==3987== I1  misses:         1,941
==3987== LLi misses:         1,871
==3987== I1  miss rate:       0.00%
==3987== LLi miss rate:       0.00%
```

versus

```
==4053== I1  misses:         1,940
==4053== LLi misses:         1,880
==4053== I1  miss rate:       0.00%
==4053== LLi miss rate:       0.00%
```

Not a lot of difference is terms of instruction cache miss.

```
==3987== D   refs:     100,773,214  (78,342,402 rd   + 22,430,812 wr)
==3987== D1  misses:       711,908  (   709,360 rd   +      2,548 wr)
==3987== LLd misses:       641,864  (   640,335 rd   +      1,529 wr)
==3987== D1  miss rate:        0.7% (      0.9%     +        0.0%  )
==3987== LLd miss rate:        0.6% (      0.8%     +        0.0%  )
```

versus

```
==4053== D   refs:     100,773,688  (78,342,700 rd   + 22,430,988 wr)
==4053== D1  misses:    11,128,717  (11,126,135 rd   +      2,582 wr)
==4053== LLd misses:    11,021,134  (11,019,573 rd   +      1,561 wr)
==4053== D1  miss rate:       11.0% (      14.2%     +        0.0%  )
==4053== LLd miss rate:       10.9% (      14.1%     +        0.0%  )
```

OK! It is pretty obvious that we are at the heart of the problem. Although the number of memory access are pretty much the same: 100M; the number of L1 and L3 misses an insanely bigger for the "row" approach.

Remember that cache misses are not free. Every one of them cost time. As "Valgrind" documentation states, a L3 cache miss can cost 200 cycles. To better understand if this is the main cause of the performance problem here. Let us analyze is my case.

```
> cat  /proc/cpuinfo|grep MHz
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
cpu MHz         : 3000.000
```

My computer has 16 cores, all of them at 3GHz. This means 3,000,000,000 cycles per second. Three billions cycles per second. Every L3 cache miss we waste 200 cycles. Does not seems much, right? But we had 11,021,134 cache misses. Eleven million. Multiplying them we get:

```
11,021,134 * 200 = 2,204,226,800 (73% of 3GHz)
```

2 billion cycles wasted. Almost a whole second of one of my CPUs!

If you sum the difference of both timings together:

```
size;colunar;row;difference
1000;0;0.001;0
10000;0;0.001;0
100000;0.005;0.013;0.008
1000000;0.055;0.129;0.074
10000000;0.546;1.371;0.825
```

Total difference is: 0.907 seconds. So from these 0.7 is basically the CPU halted waiting data from the memory because a L3 cache was missed.

But the question boils down to: why is the "row" case much worse in cache? Is there anything structural or relevant here that we must take into account? And the answer is YES. That is why so many databases choose to use a "columnar store" instead of a "row store" when the purpose of the data is to be aggregated.

DATA LAYOUT

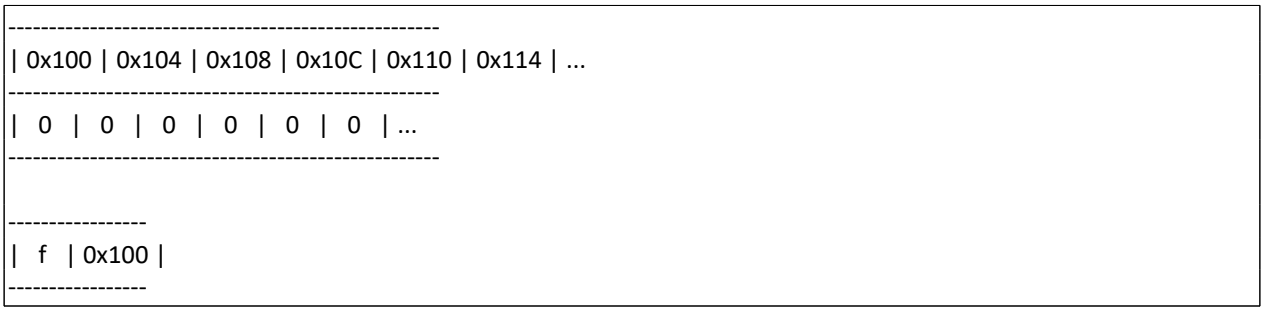
To understand the difference let us focus on the only two differences between the two cases:

```
float* f = new float[size];
```

versus

```
struct SomeData
{
   char junk[32];
   float x;
   char doesntMatter[32];
};
```

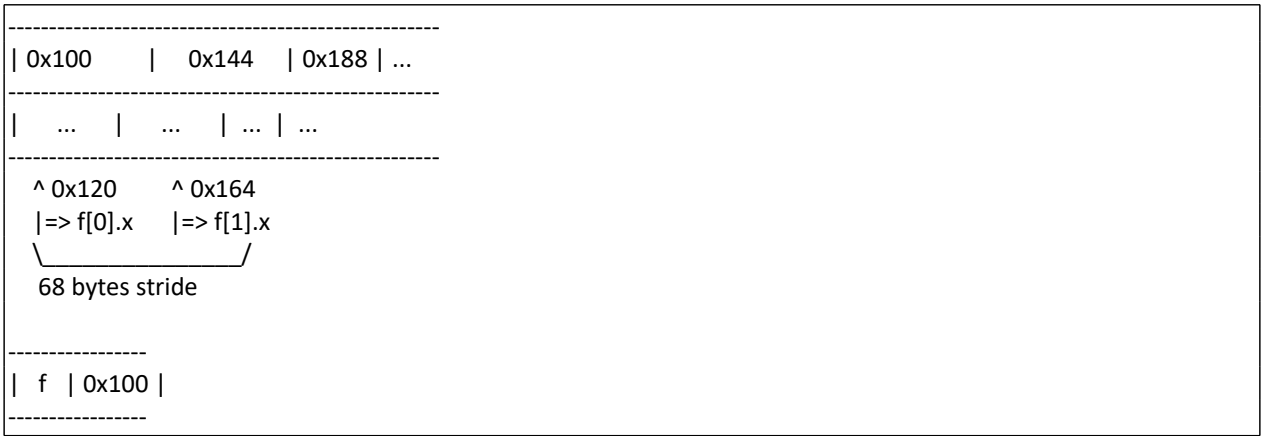
We know that c++ arrays are packed in the sense that, in this case, every float is nest to the other.



but for structs it, of course, depends on the struct. Clang has an option that allow us to see the object layout. Let us use it:

```
> clang -cc1 -fdump-record-layouts myfile.cpp

*** Dumping AST Record Layout
         0 | struct SomeData
         0 |   char [32] junk
        32 |   float x
        36 |   char [32] doesntMatter
           | [sizeof=68, dsize=68, align=4,
           |  nvsize=68, nvalign=4]
```

We can easily see that the struct takes 68 bytes, and "x" is at offset 32. So when we create an array of "SomeData" they get packed together, but the "x"s fields get apart from each other 68 bytes. This is called the stride.



This is very important here because the way the CPU cache works. In the happy case of a packed array when we access "f[0].x" the first time we will miss every cache level and we will need to bring data from memory.

Luckily, CPU engineers are smart and know that this process is very slow, in CPU time. So they bring not just the "4-bytes-you-ordered", but a chunk known as "cache line". Today "cache lines" have 64 bytes. In 64 bits computer this is 8 "words".

So the first time we read "f[0].x" we will actually bring 64 bytes from the memory to the L3 cache. In the packed array case this will automatically give us:

So even if the "cache predictor" do nothing, we will gain "f[1]" to "f[7]" in the cache for free. Because of the "cache line" strategy. Now compare this with the struct.

The struct itself is bigger than the "cache line". So the first time I read "f[0].x", even bringing 64 more bytes, "f[1].x" will not be in the cache. So I will have automatically a cache miss in the next iteration.

Let us use a simple cost-of-access to measure the difference. Everything is free but:

1 - Cache access cost 10

2 - Memory access cost 200

This is not even close to the truth, but will do for now.

Packed case:

```
Read f[0] - 200
Read f[1] - 10
Read f[2] - 10
Read f[3] - 10
Read f[4] - 10
Read f[5] - 10
Read f[6] - 10
Read f[7] - 10
Read f[8] - 200
Read f[9] – 10
...
```

For the first 10 we have: 480.

Struct casse:

```
Read f[0].x - 200
Read f[1].x - 200
Read f[2].x - 200
Read f[3].x - 200
Read f[4].x - 200
Read f[5].x - 200
Read f[6].x - 200
Read f[7].x - 200
Read f[8].x - 200
Read f[9].x - 200
...
```

For the first 10, we have: 2000. Five times more.

And actually get worse. Why? Cache prefetching. This means that the CPU will try to anticipate you memory read pattern and will bring data from memory to the cache before you ask.

Data cache prefetching is extremely difficult is various cases but one: linear access. So in this case, suppose the "prefetcher" suspects that we will read the chunk. We will have something like this.

```
Read f[0] - 200 - prefetcher reads next 8 bytes will take 200
Read f[1] - 10
Read f[2] - 10
Read f[3] - 10
Read f[4] - 10
Read f[5] - 10
Read f[6] - 10
Read f[7] - 10
Read f[8] - 130 - the remaining from the prefetcher read
Read f[9] - 10
...
```

versus

```
Read f[0].x - 200 - prefetcher reads next 8 bytes will take 200
Read f[1].x - 200 - outside prefetcher guess. No effect
Read f[2].x - 200
Read f[3].x - 200
Read f[4].x - 200
Read f[5].x - 200
Read f[6].x - 200
Read f[7].x - 200
Read f[8].x - 200
Read f[9].x - 200
...
```

So we have another advantage to having packed data. The "cache prefetcher" will anticipate our use.

PREFETCH BUILTIN

In this simplistic scenario, we the "for" body is very small is hard to improve the situation of the "row" case. But one can always use some form of compiler intrinsic builtin to help the "cache prefetcher".

> — Built-in Function: void __builtin_prefetch (const void *addr, ...)
>
> This function is used to minimize cache-miss latency by moving data
>
> into a cache before it is accessed.

https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/Other-Builtins.html

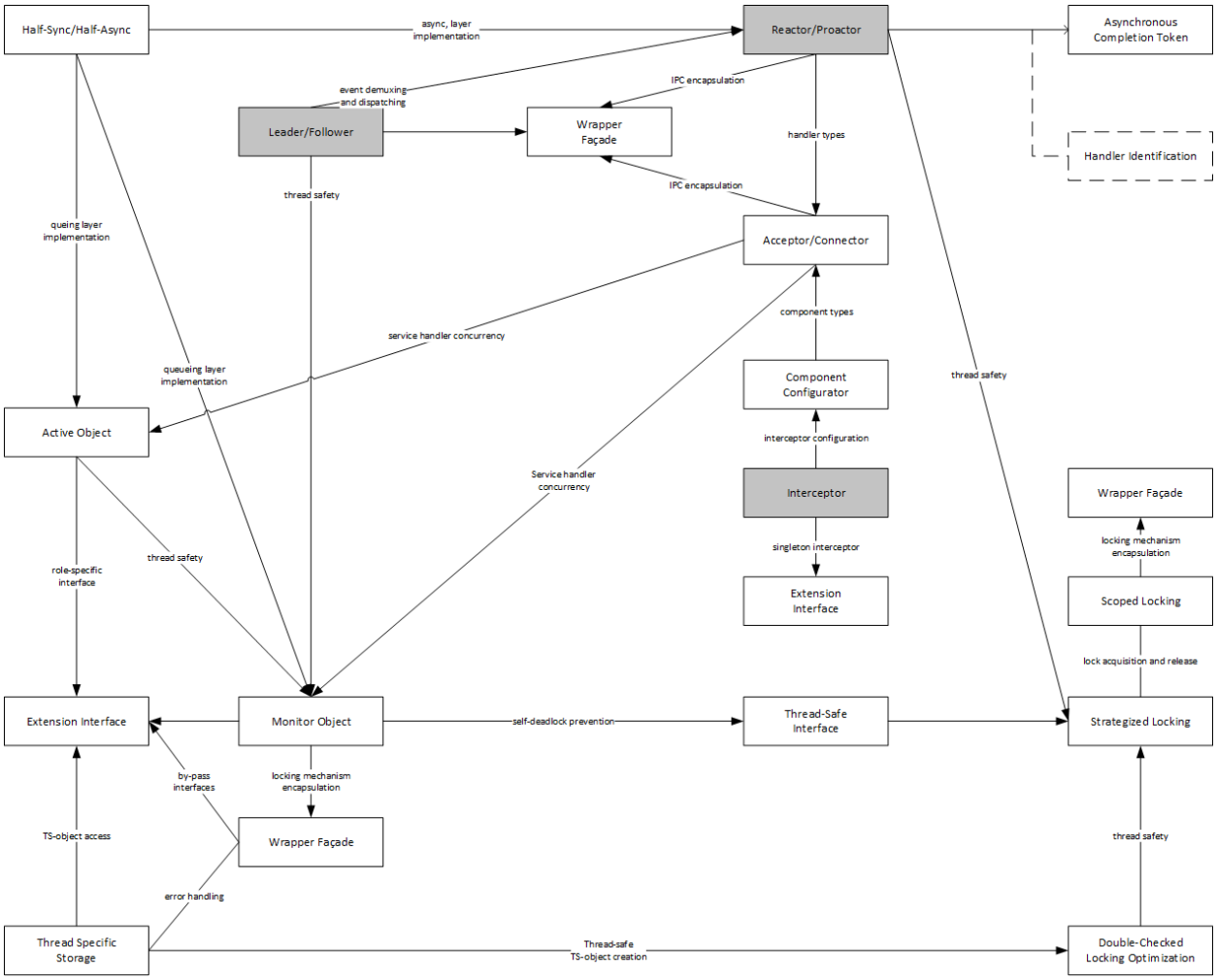
```
SomeData* arr = new SomeData[size];

float accum = 0;
for(int i = 0; i < size; ++i)
{
   accum += arr[i].x;
   __builtin_prefetch(&(arr[i+1].x), 1);
}

delete[] arr;
```

In this case this not only does not improve the performance, but can actually decrease, because the calculation of the next address is almost as costly as the calculation we do inside the for, doubling our work.

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:

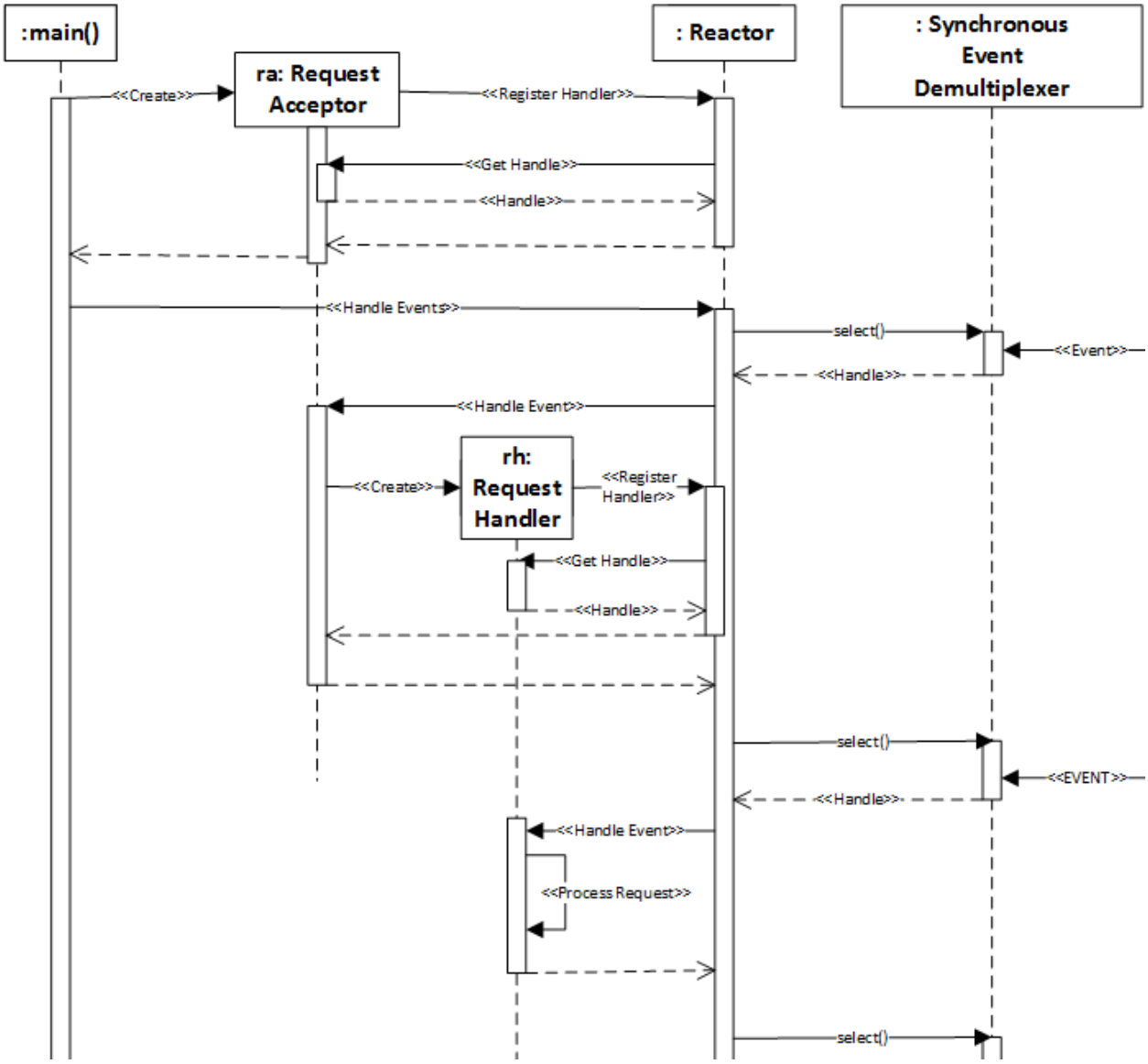
Pattern Hatching: Design Patterns Applied

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.

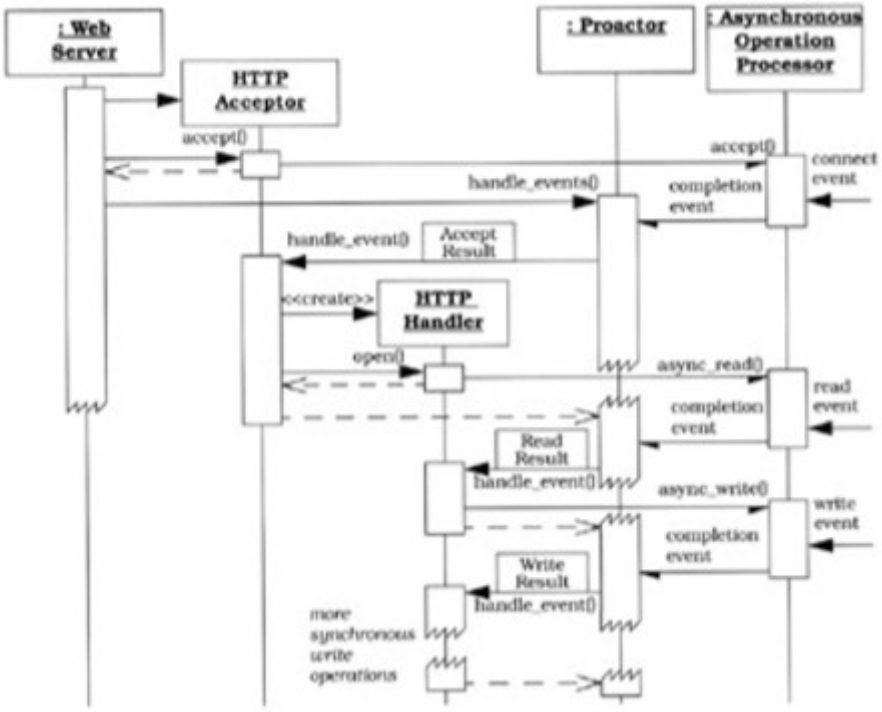
The Proactor architectural pattern allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations. It offers the performance benefits of concurrency without incurring some of its liabilities

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).

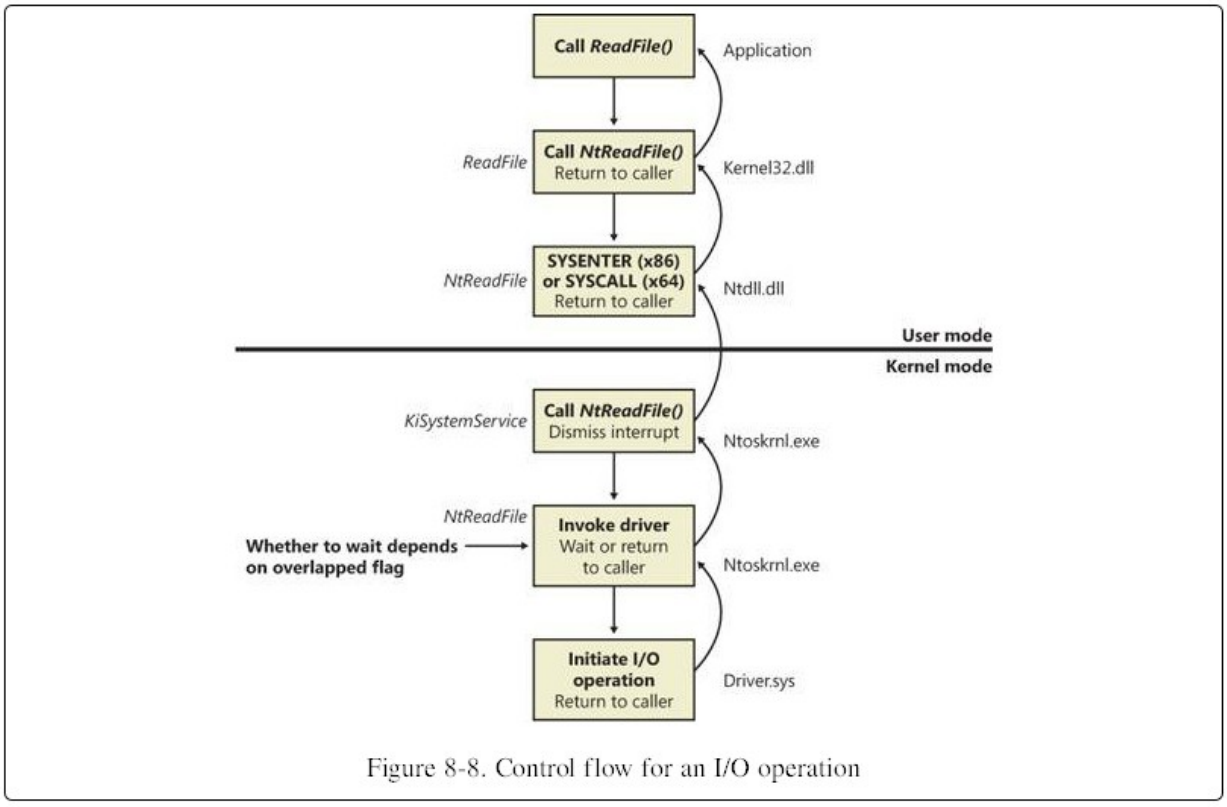


Figure 8-8. Control flow for an I/O operation

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.

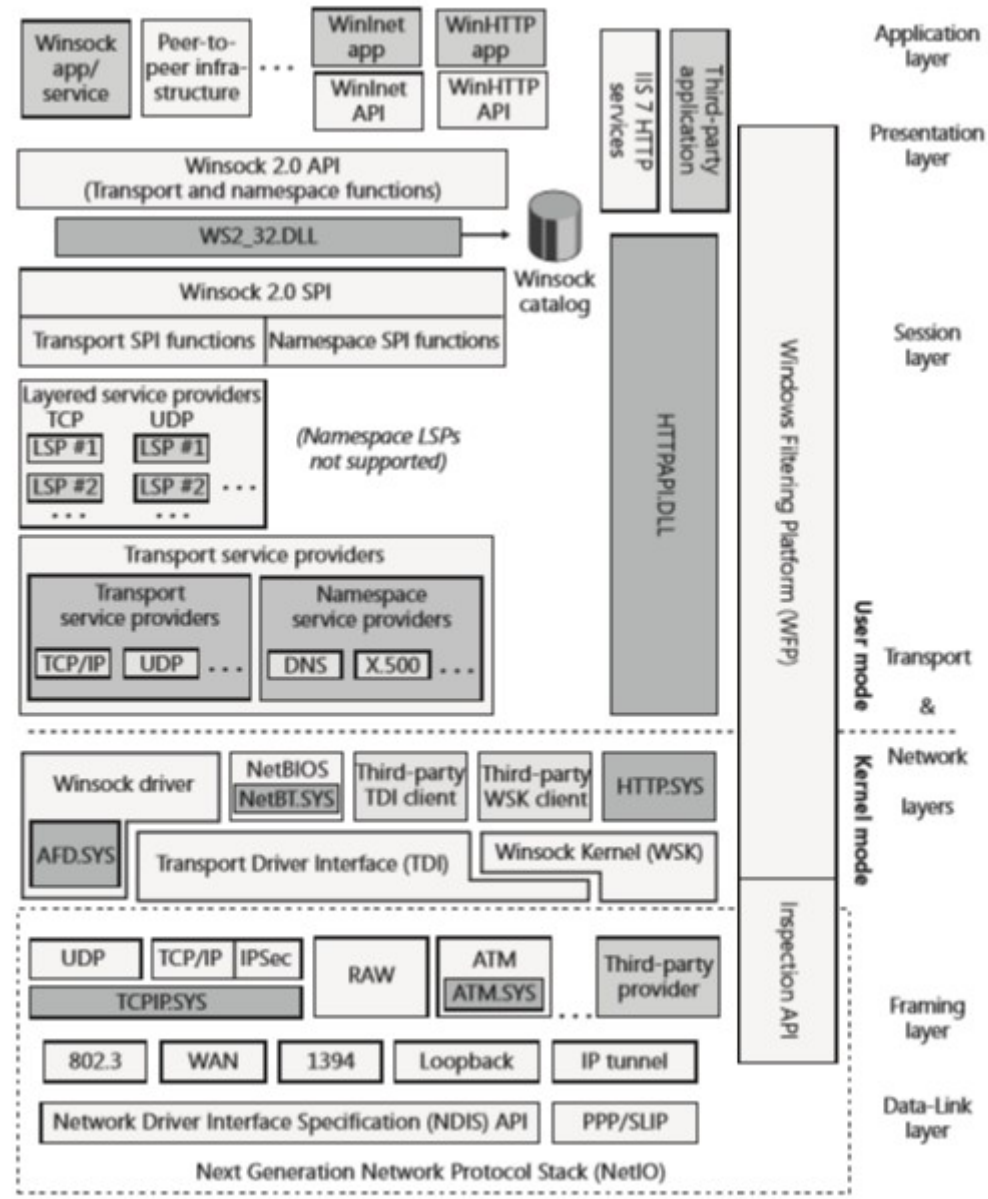


FIGURE 12-2 OSI model and Windows networking components

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

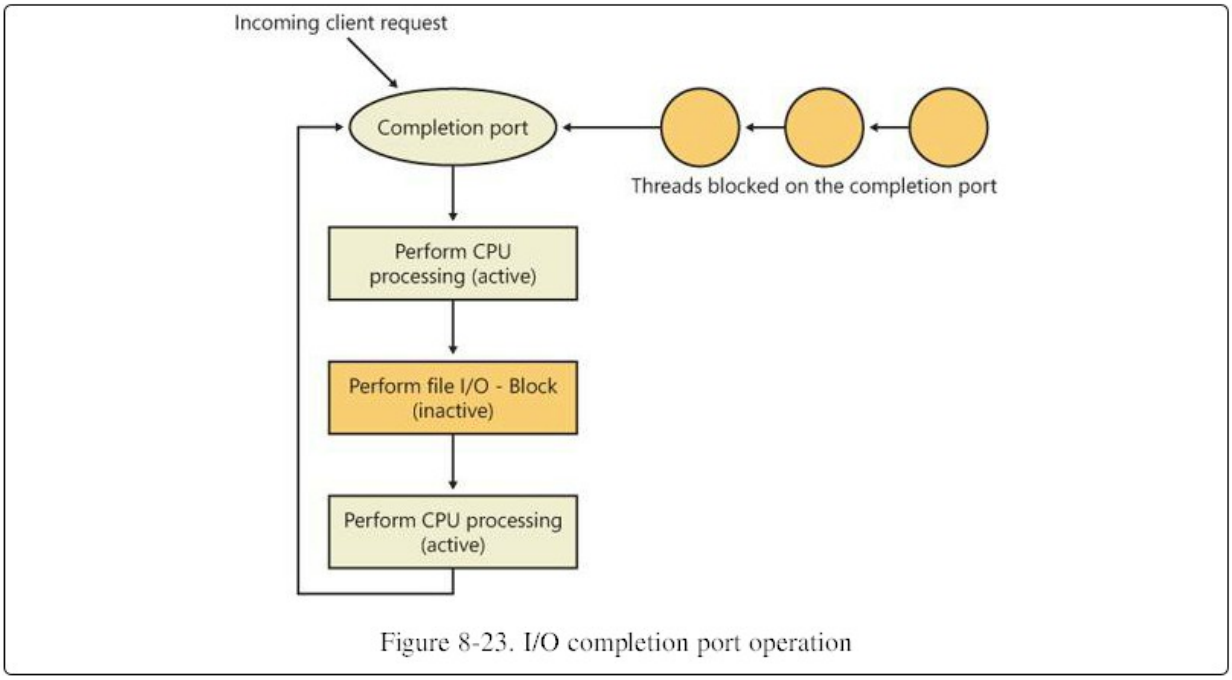


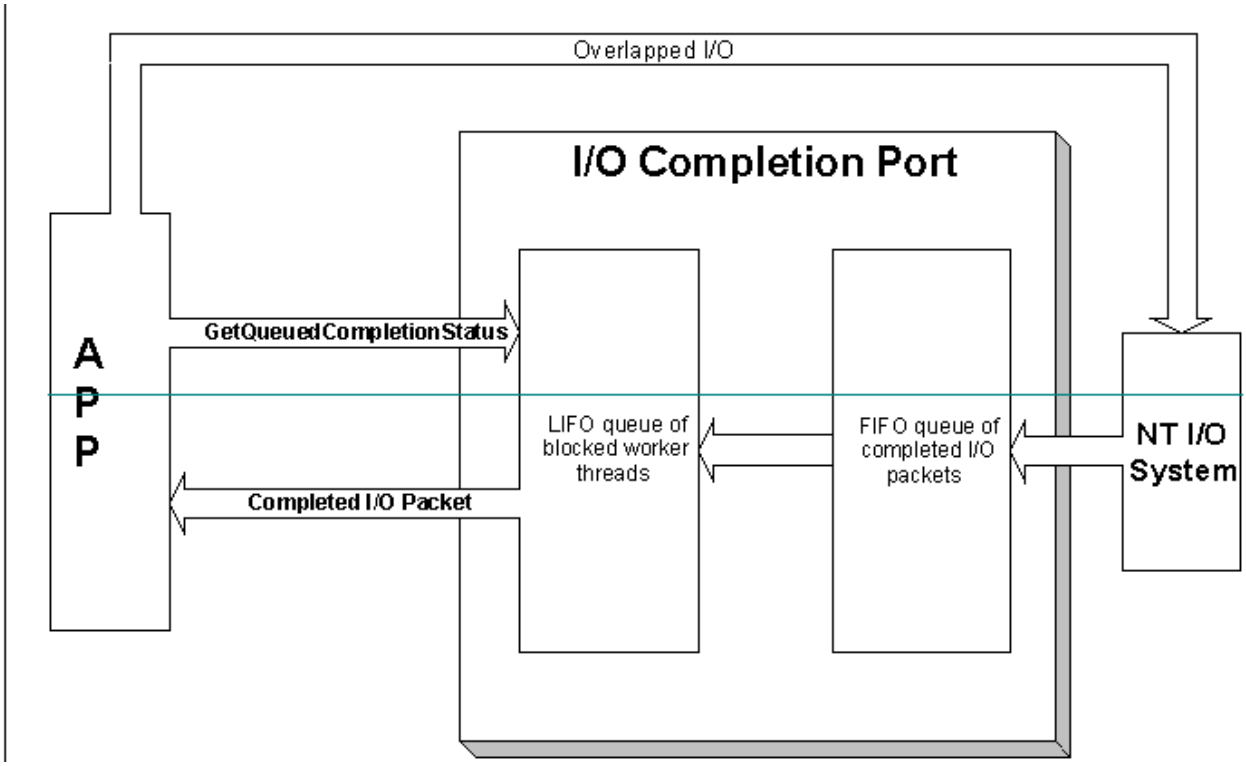
Figure 8-23. I/O completion port operation

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

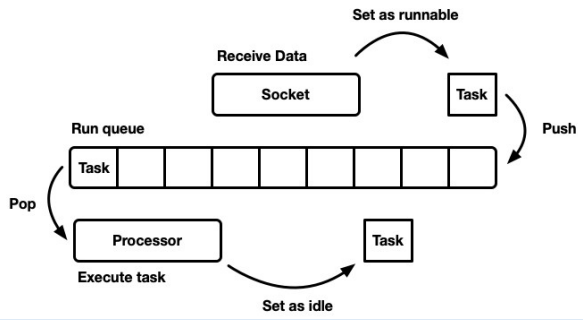
SCHEDULER ARCHITECTURE

https://en.wikipedia.org/wiki/Thread_(computing)#M:N_(hybrid_threading)

https://stackoverflow.com/questions/5004162/what-does-it-mean-for-a-data-structure-to-be-intrusive

https://tokio.rs/blog/2019-10-scheduler/

http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue



SIMPLES IMPLEMENTATION

```
auto q = std::queue<Task>();

while(q.NotEmpty()){

auto t = q.unqueue();

t.run();

}
```

ONE QUEUE, MANY PROCESSORS

MANY QUEUES, MANY PROCESSORS

SYNCHRONIZED QUEUE

TWO QUEUES: PUBLIC SYNC, PRIVATE NON-SYNC

THREAD-SAFE PUSH QUEUE (MPSC)

3 – ALGORITHMS

https://github.com/crossbeam-rs/crossbeam - single-producer, multi-consumer deque

https://www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf

[TODO]

thundering herd problem

atomic memory orderings

4 – TESTING

[TODO]

https://users.soe.ucsc.edu/~cormac/papers/popl05.pdf