

## Introduction

The purpose of this document is to share some ideas that I've developed over the years about how to develop a certain kind of application for which the term "server" is only a weak approximation. More accurately, I'll be writing about a broad class of programs that are designed to handle very large numbers of discrete messages or requests per second. Network servers most commonly fit this definition, but not all programs that do are really servers in any sense of the word. For the sake of simplicity, though, and because "High-Performance Request-Handling Programs" is a really lousy title, we'll just say "server" and be done with it.

I will not be writing about "mildly parallel" applications, even though multitasking within a single program is now commonplace. The browser you're using to read this probably does some things in parallel, but such low levels of parallelism really don't introduce many interesting challenges. The interesting challenges occur when the request-handling infrastructure itself is the limiting factor on overall performance, so that improving the infrastructure actually improves performance. That's not often the case for a browser running on a gigahertz processor with a gigabyte of memory doing six simultaneous downloads over a DSL line. The focus here is not on applications that sip through a straw but on those that drink from a firehose, on the very edge of hardware capabilities where how you do it really does matter.

Some people will inevitably take issue with some of my comments and suggestions, or think they have an even better way. Fine. I'm not trying to be the Voice of God here; these are just methods that I've found to work for me, not only in terms of their effects on performance but also in terms of their effects on the difficulty of debugging or extending code later. Your mileage may vary. If something else works better for you that's great, but be warned that almost everything I suggest here exists as an alternative to something else that I tried once only to be disgusted or horrified by the results. Your pet idea might very well feature prominently in one of these stories, and innocent readers might be bored to death if you encourage me to start telling them. You wouldn't want to hurt them, would you?

The rest of this article is going to be centered around what I'll call the Four Horsemen of Poor Performance:

1. Data copies
2. Context switches
3. Memory allocation
4. Lock contention





There will also be a catch-all section at the end, but these are the biggest performance-killers. If you can handle most requests without copying data, without a context switch, without going through the memory allocator and without contending for locks, you'll have a server that performs well even if it gets some of the minor parts wrong.

## Data Copies

This could be a very short section, for one very simple reason: most people have learned this lesson already. Everybody knows data copies are bad; it's obvious, right? Well, actually, it probably only seems obvious because you learned it very early in your computing career, and that only happened because somebody started putting out the word decades ago. I know that's true for me, but I digress. Nowadays it's covered in every school curriculum and in every informal how-to. Even the marketing types have figured out that "zero copy" is a good buzzword.

Despite the after-the-fact obviousness of copies being bad, though, there still seem to be nuances that people miss. The most important of these is that data copies are often hidden and disguised. Do you really know whether any code you call in drivers or libraries does data copies? It's probably more than you think. Guess what "Programmed I/O" on a PC refers to. An example of a copy that's disguised rather than hidden is a hash function, which has all the memory-access cost of a copy and also involves more computation. Once it's pointed out that hashing is effectively "copying plus" it seems obvious that it should be avoided, but I know at least one group of brilliant people who had to figure it out the hard way. If you really want to get rid of data copies, either because they really are hurting performance or because you want to put "zero-copy operation" on your hacker-conference slides, you'll need to track down a lot of things that really are data copies but don't advertise themselves as such.

The tried and true method for avoiding data copies is to use indirection, and pass buffer descriptors (or chains of buffer descriptors) around instead of mere buffer pointers. Each descriptor typically consists of the following:

-  A pointer and length for the whole buffer.
-  A pointer and length, or offset and length, for the part of the buffer that's actually filled.
-  Forward and back pointers to other buffer descriptors in a list.
-  A reference count.

Now, instead of copying a piece of data to make sure it stays in memory, code can simply increment a reference count on the appropriate buffer descriptor. This can work extremely well under some conditions, including the way that a typical network protocol stack operates, but it can also become a really big headache. Generally speaking, it's easy to add buffers at the beginning or end of a chain, to add references to whole buffers, and to deallocate a whole chain at once. Adding in the middle, deallocating piece by piece, or referring to partial buffers will each make life increasingly difficult. Trying to split or combine buffers will simply drive you insane.

I don't actually recommend using this approach for everything, though. Why not? Because it gets to be a huge pain when you have to walk through descriptor chains every time you want to look at a header field. There really are worse things than data copies. I find that the best thing to do is to identify the large objects in a program, such as data blocks, make sure those get allocated separately as described above so that they don't need to be copied, and not sweat too much about the other stuff.

This brings me to my last point about data copies: don't go overboard avoiding them. I've seen way too much code that avoids data copies by doing something even worse, like forcing a context switch or breaking up a large I/O request. Data copies are expensive, and when you're looking for places to avoid redundant operations they're one of the first things you should look at, but there is a point of diminishing returns. Combing through code and then making it twice as complicated just to get rid of that last few data copies is usually a waste of time that could be better spent in other ways.

## Context Switches




Whereas everyone thinks it's obvious that data copies are bad, I'm often surprised by how many people totally ignore the effect of context switches on performance. In my experience, context switches are actually behind more total "meltdowns" at high load than data copies: the system starts spending more time going from one thread to another than it actually spends within any thread doing useful work. The amazing thing is that, at one level, it's totally obvious what causes excessive context switching. The #1 cause of context switches is having more active threads than you have processors. As the ratio of active threads to processors increases, the number of context switches also increases - linearly if you're lucky, but often exponentially. This very simple fact explains why multi-threaded designs that have one thread per connection scale very poorly. The only realistic alternative for a scalable system is to limit the number of active threads so it's (usually) less than or equal to the number of processors. One popular variant of this approach is to use only one thread, ever; while such an approach does avoid context thrashing, and avoids the need for locking as well, it is also incapable of achieving more than one processor's worth of total throughput and thus remains beneath contempt unless the program will be non-CPU-bound (usually network-I/O-bound) anyway.

The first thing that a "thread-frugal" program has to do is figure out how it's going to make one thread handle multiple connections at once. This usually implies a front end that uses select/poll, asynchronous I/O, signals or completion ports, with an event-driven structure behind that. Many "religious wars" have been fought, and continue to be fought, over which of the various front-end APIs is best. Dan Kegel's [C10K paper](#) is a good resource in this area. Personally, I think all flavors of select/poll and signals are ugly hacks, and therefore favor either AIO or completion ports, but it actually doesn't matter that much. They all - except maybe select() - work reasonably well, and don't really do much to address the matter of what happens past the very outermost layer of your program's front end.

The simplest conceptual model of a multi-threaded event-driven server has a queue at its center; requests are read by one or more "listener" threads and put on queues, from which one or more "worker" threads will remove and process them. Conceptually, this is a good model, but all too often people actually implement their code this way. Why is this wrong? Because the #2 cause of context switches is transferring work from one thread to another. Some people even compound the error by requiring that the response to a request be sent by the original thread - guaranteeing not one but two context switches per request. It's very important to use a "symmetric" approach in which a given thread can go from being a listener to a worker to a listener again without ever changing context. Whether this involves partitioning connections between threads or having all threads take turns being listener for the entire set of connections seems to matter a lot less.

Usually, it's not possible to know how many threads will be active even one instant into the future. After all, requests can come in on any connection at any moment, or "background" threads dedicated to various maintenance tasks could pick that moment to wake up. If you don't know how many threads are active, how can you limit how many are active? In my experience, one of the most effective approaches is also one of the simplest: use an old-fashioned counting semaphore which each thread must hold whenever it's doing "real work". If the thread limit has already been reached then each listen-mode thread might incur one extra context switch as it wakes up and then blocks on the semaphore, but once all listen-mode threads have blocked in this way they won't continue contending for resources until one of the existing threads "retires" so the system effect is negligible. More importantly, this method handles maintenance threads - which sleep most of the time and therefore don't count against the active thread count - more gracefully than most alternatives.

Once the processing of requests has been broken up into two stages (listener and worker) with multiple threads to service the stages, it's natural to break up the processing even further into more than two stages. In its simplest form, processing a request thus becomes a matter of invoking stages successively in one direction, and then in the other (for replies). However, things can get more complicated; a stage might represent a "fork" between two processing paths which involve different stages, or it might generate a reply (e.g. a cached value) itself without invoking further stages. Therefore, each stage needs to be able to specify "what should happen next" for a request. There are three possibilities, represented by return values from the stage's dispatch function:

-  The request needs to be passed on to another stage (an ID or pointer in the return value).
-  The request has been completed (a special "request done" return value)
-  The request was blocked (a special "request blocked" return value). This is equivalent to the previous case, except that the request is not freed and will be continued later from another thread.

Note that, in this model, queuing of requests is done within stages, not between stages. This avoids the common silliness of constantly putting a request on a successor stage's queue, then immediately invoking that successor stage and dequeuing the request again; I call that lots of queue activity - and locking - for nothing.

If this idea of separating a complex task into multiple smaller communicating parts seems familiar, that's because it's actually very old. My approach has its roots in the [Communicating Sequential Processes](#) concept elucidated by C.A.R. Hoare in 1978, based in turn on ideas from Per Brinch Hansen and Matthew Conway going back to 1963 - before I was born! However, when Hoare coined the term CSP he meant "process" in the abstract mathematical sense, and a CSP process need bear no relation to the operating-system entities of the same name. In my opinion, the common approach of implementing CSP via thread-like coroutines within a single OS thread gives the user all of the headaches of concurrency with none of the scalability.

A contemporary example of the staged-execution idea evolved in a saner direction is Matt Welsh's [SEDA](#). In fact, SEDA is such a good example of "server architecture done right" that it's worth commenting on some of its specific characteristics (especially where those differ from what I've outlined above).

1. SEDA's "batching" tends to emphasize processing multiple requests through a stage at once, while my approach tends to emphasize processing a single request through multiple stages at once.
2. SEDA's one significant flaw, in my opinion, is that it allocates a separate thread pool to each stage with only "background" reallocation of threads between stages in response to load. As a result, the #1 and #2 causes of context switches noted above are still very much present.
3. In the context of an academic research project, implementing SEDA in Java might make sense. In the real world, though, I think the choice can be characterized as unfortunate.

## Memory Allocation

Allocating and freeing memory is one of the most common operations in many applications. Accordingly, many clever tricks have been developed to make general-purpose memory allocators more efficient. However, no amount of cleverness can make up for the fact that the very generality of such allocators inevitably makes them far less efficient than the alternatives in many cases. I therefore have three suggestions for how to avoid the system memory allocator altogether.

Suggestion #1 is simple preallocation. We all know that static allocation is bad when it imposes artificial limits on program functionality, but there are many other forms of preallocation that can be quite beneficial. Usually the reason comes down to the fact that one trip through the system memory allocator is better than several, even when some memory is "wasted" in the process. Thus, if it's possible to assert that no more than  $N$  items could ever be in use at once, preallocation at program startup might be a valid choice. Even when that's not the case, preallocating everything that a request handler might need right at the beginning might be preferable to allocating each piece as it's needed; aside from the possibility of allocating multiple items contiguously in one trip through the system allocator, this often greatly simplifies error-recovery code. If memory is very tight then preallocation might not be an option, but in all but the most extreme circumstances it generally turns out to be a net win.

Suggestion #2 is to use lookaside lists for objects that are allocated and freed frequently. The basic idea is to put recently-freed objects onto a list instead of actually freeing them, in the hope that if they're needed again soon they need merely be taken off the list instead of being allocated from system memory. As an additional benefit, transitions to/from a lookaside list can often be implemented to skip complex object initialization/finalization.

It's generally undesirable to have lookaside lists grow without bound, never actually freeing anything even when your program is idle. Therefore, it's usually necessary to have some sort of periodic "sweeper" task to free inactive objects, but it would also be undesirable if the sweeper introduced undue locking complexity or contention. A good compromise is therefore a system in which a lookaside list actually consists of separately locked "old" and "new" lists. Allocation is done preferentially from the new list, then from the old list, and from the system only as a last resort; objects are always freed onto the new list. The sweeper thread operates as follows:

1. Lock both lists.
2. Save the head for the old list.
3. Make the (previously) new list into the old list by assigning list heads.
4. Unlock.
5. Free everything on the saved old list at leisure.

Objects in this sort of system are only actually freed when they have not been needed for at least one full sweeper interval, but always less than two. Most importantly, the sweeper does most of its work without holding any locks to contend with regular threads. In theory, the same approach can be generalized to more than two stages, but I have yet to find that useful.

One concern with using lookaside lists is that the list pointers might increase object size. In my experience, most of the objects that I'd use lookaside lists for already contain list pointers anyway, so it's kind of a moot point. Even if the pointers were only needed for the lookaside lists, though, the savings in terms of avoided trips through the system memory allocator (and object initialization) would more than make up for the extra memory.

Suggestion #3 actually has to do with locking, which we haven't discussed yet, but I'll toss it in anyway. Lock contention is often the biggest cost in allocating memory, even when lookaside lists are in use. One solution is to maintain multiple private lookaside lists, such that there's absolutely no possibility of contention for any one list. For example, you could have a separate lookaside list for each thread. One list per processor can be even better, due to cache-warmth considerations, but only works if threads cannot be preempted. The private lookaside lists can even be combined with a shared list if necessary, to create a system with extremely low allocation overhead.



#### Lock Contention

Efficient locking schemes are notoriously hard to design, because of what I call Scylla and Charybdis after the monsters in the Odyssey. Scylla is locking that's too simplistic and/or coarse-grained, serializing activities that can or should proceed in parallel and thus sacrificing performance and scalability; Charybdis is overly complex or fine-grained locking, with space for locks and time for lock operations again sapping performance. Near Scylla are shoals representing deadlock and livelock conditions; near Charybdis are shoals representing race conditions. In between, there's a narrow channel that represents locking which is both efficient and correct...or is there? Since locking tends to be deeply tied to program logic, it's often impossible to design a good locking scheme without fundamentally changing how the program works. This is why people hate locking, and try to rationalize their use of non-scalable single-threaded approaches.

Almost every locking scheme starts off as "one big lock around everything" and a vague hope that performance won't suck. When that hope is dashed, and it almost always is, the big lock is broken up into smaller ones and the prayer is repeated, and then the whole process is repeated, presumably until performance is adequate. Often, though, each iteration increases complexity and locking overhead by 20-50% in return for a 5-10% decrease in lock contention. With luck, the net result is still a modest increase in performance, but actual decreases are not uncommon. The designer is left scratching his head (I use "his" because I'm a guy myself; get over it). "I made the locks finer grained like all the textbooks said I should," he thinks, "so why did performance get worse?"




In my opinion, things got worse because the aforementioned approach is fundamentally misguided. Imagine the "solution space" as a mountain range, with high points representing good solutions and low points representing bad ones. The problem is that the "one big lock" starting point is almost always separated from the higher peaks by all manner of valleys, saddles, lesser peaks and dead ends. It's a classic hill-climbing problem; trying to get from such a starting point to the higher peaks only by taking small steps and never going downhill almost never works. What's needed is a fundamentally different way of approaching the peaks.

The first thing you have to do is form a mental map of your program's locking. This map has two axes:

-  The vertical axis represents code. If you're using a staged architecture with non-branching stages, you probably already have a diagram showing these divisions, like the ones everybody uses for OSI-model network protocol stacks.
-  The horizontal axis represents data. In every stage, each request should be assigned to a data set with its own resources separate from any other set.

You now have a grid, where each cell represents a particular data set in a particular processing stage. What's most important is the following rule: two requests should not be in contention unless they are in the same data set and the same processing stage. If you can manage that, you've already won half the battle.






Once you've defined the grid, every type of locking your program does can be plotted, and your next goal is to ensure that the resulting dots are as evenly distributed along both axes as possible. Unfortunately, this part is very application-specific. You have to think like a diamond-cutter, using your knowledge of what the program does to find the natural "cleavage lines" between stages and data sets. Sometimes they're obvious to start with. Sometimes they're harder to find, but seem more obvious in retrospect. Dividing code into stages is a complicated matter of program design, so there's not much I can offer there, but here are some suggestions for how to define data sets:

-  If you have some sort of a block number or hash or transaction ID associated with requests, you can rarely do better than to divide that value by the number of data sets.
-  Sometimes, it's better to assign requests to data sets dynamically, based on which data set has the most resources available rather than some intrinsic property of the request. Think of it like multiple integer units in a modern CPU; those guys know a thing or two about making discrete requests flow through a system.
-  It's often helpful to make sure that the data-set assignment is different for each stage, so that requests which would contend at one stage are guaranteed not to do so at another stage.

If you've divided your "locking space" both vertically and horizontally, and made sure that lock activity is spread evenly across the resulting cells, you can be pretty sure that your locking is in pretty good shape. There's one more step, though. Do you remember the "small steps" approach I derided a few paragraphs ago? It still has its place, because now you're at a good starting point instead of a terrible one. In metaphorical terms you're probably well up the slope on one of the mountain range's highest peaks, but you're probably not at the top of one. Now is the time to collect contention statistics and see what you need to do to improve, splitting stages and data sets in different ways and then collecting more statistics until you're satisfied. If you do all that, you're sure to have a fine view from the mountaintop.

#### Other Stuff

As promised, I've covered the four biggest performance problems in server design. There are still some important issues that any particular server will need to address, though. Mostly, these come down to knowing your platform/environment:

-  How does your storage subsystem perform with larger vs. smaller requests? With sequential vs. random? How well do read-ahead and write-behind work?
-  How efficient is the network protocol you're using? Are there parameters or flags you can set to make it perform better? Are there facilities like TCP\_CORK, MSG\_PUSH, or the Nagle-toggling trick that you can use to avoid tiny messages?
-  Does your system support scatter/gather I/O (e.g. readv/writv)? Using these can improve performance and also take much of the pain out of using buffer chains.
-  What's your page size? What's your cache-line size? Is it worth it to align stuff on these boundaries? How expensive are system calls or context switches, relative to other things?
-  Are your reader/writer lock primitives subject to starvation? Of whom? Do your events have "thundering herd" problems? Does your sleep/wakeup have the nasty (but very common) behavior that when X wakes Y a context switch to Y happens immediately even if X still has things to do?

I'm sure I could think of many more questions in this vein. I'm sure you could too. In any particular situation it might not be worthwhile to do anything about any one of these issues, but it's usually worth at least thinking about them. If you don't know the answers - many of which you will not find in the system documentation - find out. Write a test program or micro-benchmark to find the answers empirically; writing such code is a useful skill in and of itself anyway. If you're writing code to run on multiple platforms, many of these questions correlate with points where you should probably be abstracting functionality into per-platform libraries so you can realize a performance gain on that one platform that supports a particular feature.

The "know the answers" theory applies to your own code, too. Figure out what the important high-level operations in your code are, and time them under different conditions. This is not quite the same as traditional profiling; it's about measuring design elements, not actual implementations. Low-level optimization is generally the last resort of someone who screwed up the design.