

DevFest
Siberia 2017

Performance without the event loop

David Cheney
heptio

Hello!

My name is David, I've come all the way from Australia to talk to you today about Go, the programming language.

I'm extremely excited to be here. So thank you to Leo, Ed, Elena, and the rest of the organisers for allowing me to be here today, it's truly a humbling moment for me.

I want to ask you two questions

But, before we begin, I want to ask you the audience, two questions

Are computers getting faster?

The first question is, are computers getting faster?

Depending on who you talk to people will either say “no, computers stopped getting faster some time ago”, or they may say “yes, of course computers are still getting faster”.

So, can I ask you to raise your hand if you believe that in general that computers are *still* getting faster, compared to a year, or maybe even five years ago.

Ok, that's a promising number.

*Do we need to rewrite our
programs to make them faster?*

ok, so the second question

Can I ask, if you have your hand raised, do you think that as a programmer, you can take advantage of the factors that continue to make computer faster *without* rewriting your programs.

In other words, will the programs that were written in the past continue to get faster *for free*?

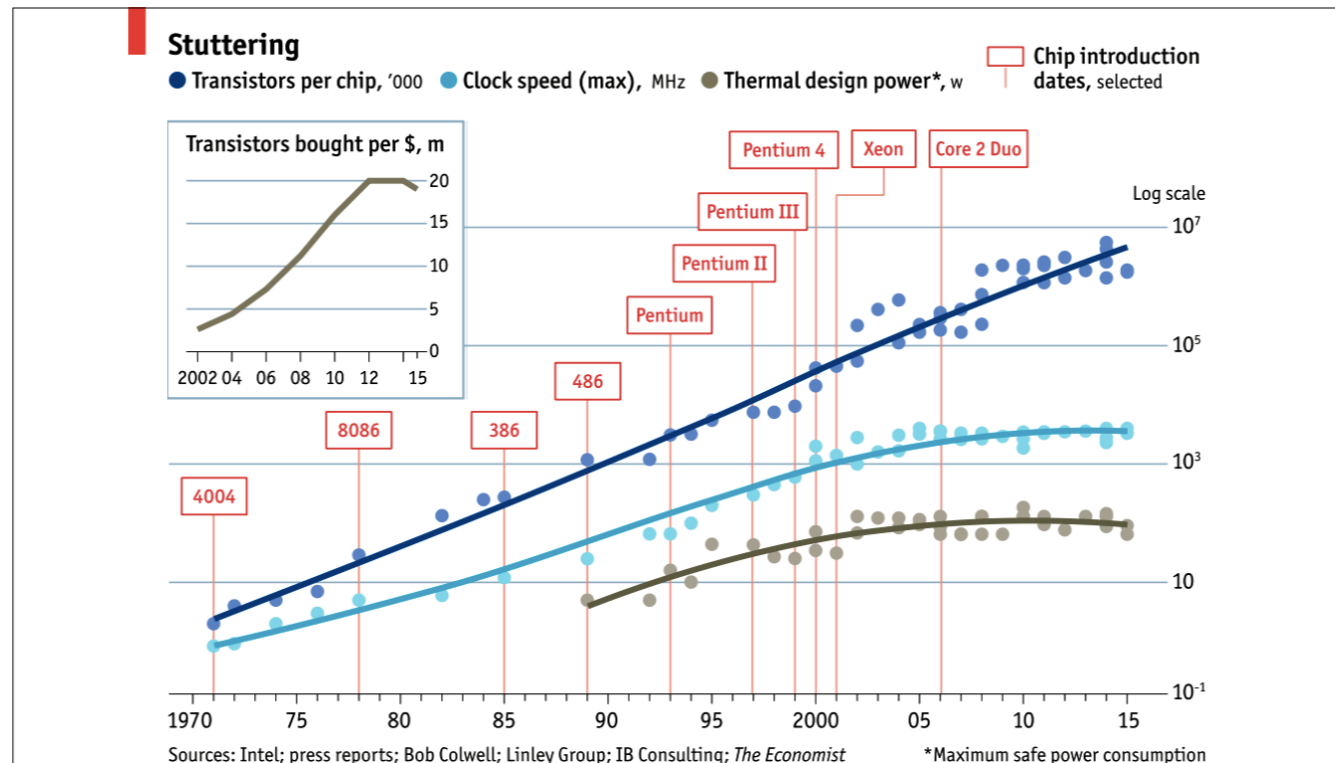
*Before we talk about fast code
we need to talk about the hardware
that will execute this code*

The title of this talk is “performance without the event loop”, but before we can talk about writing high performance code, we need to talk about the hardware that will execute this code.

As software authors all of us in this room have benefited from Moore's Law, the doubling of the number of available transistors on a chip every 18 months, for 50 years.

No other industry has experienced a six order of magnitude improvement in their tools in the space of a lifetime.

But this is all changing.



In this graph we can see that the number of transistors per CPU die continues to double roughly every 18-24 months

However if we look at the middle graph, we see clock speeds have not increased in a decade, we see that cpu speeds stalled around 2007

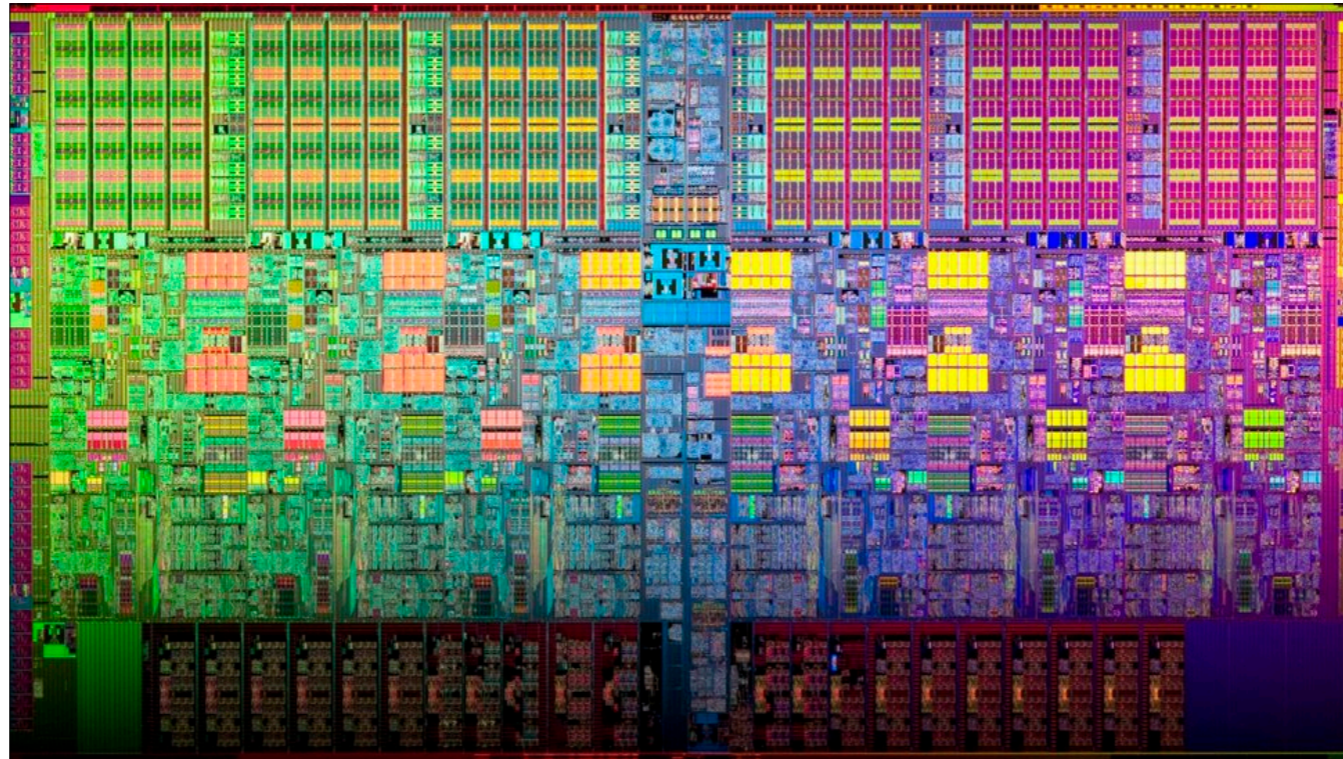
(click)

More worrying, and the reason that I put this graph here, is the insert, the number of transistors purchased per dollar. That number hit a wall around 2012 and now is starting to fall — a dollar now buys less transistors than it did 5 years ago. Why is this?

CPU manufacturers have continued to pack more transistors on the same sized cpu die by making them smaller.

But as each reduction in transistor size occurs it is costing intel, Transmeta, AMD, and Samsung billions of dollars because they have to build new fabs, buy all new process tooling. So while the number of transistors per die continues to increase, their unit cost has started to increase.

So, what are most of these transistors doing?



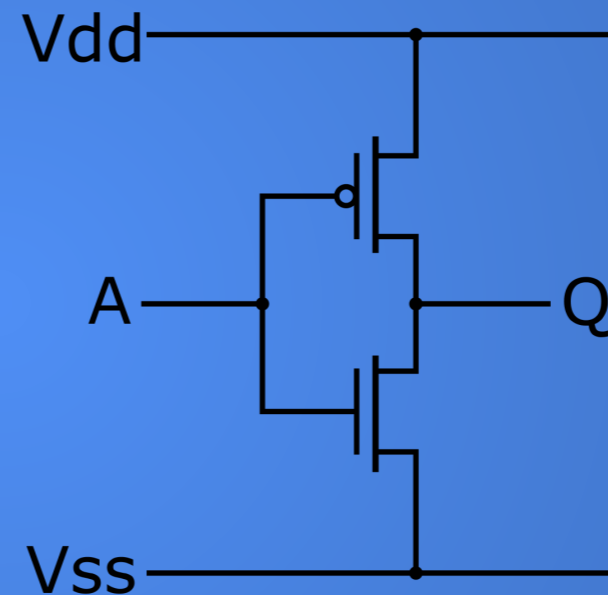
They're going towards adding more CPU cores.

CPUs are not getting faster, but they are getting wider with hyper threading and multiple cores. Dual core on mobile parts, quad core on desktop parts, dozens of cores on server parts.

In truth, the core count of a CPU is dominated by heat dissipation. So much so that the clock speed of a CPU is some arbitrary number between 1 and 4 Ghz depending on how hot the CPU is.

So, why is heat the problem?

Complementary Metal Oxide Semiconductor (CMOS)



This is a schematic of the simplest logic gate inside the CPU, an inverter — if A is high, Q will be low, if A is low, Q is high.

This gate consists of two transistors built out of a process called Complementary Metal Oxide Semiconductor, or CMOS for short. This is the process that is used for every consumer CPU in this room.

The complementary part is the key. In a steady state, when A is either high, or low, no current flows directly from the source to the drain.

However, during transitions there is a brief period where both transistors are conducting there is effectively a dead short between source and drain. Now this transition occurs very quickly, pico seconds, but when you have *billions* of transistors across the CPU doing this billions of times per second, gigahertz, all these little shorts add up to a lot of current dissipated as heat.

Power consumption of a chip, and thus heat dissipation, is directly proportional to major two factors

1. number of transition per second—Clock speed
2. gate leakage. these two transistors are not ideal, even in steady state, one is never perfectly off, and the other is never perfectly on, so they both behave like resistors. This leakage represents a static power drain, burnt off as heat.

Smaller transistors are aimed at reducing power consumption not improving performance.

So, now we know that CPU feature size reductions are primarily aimed at reducing power consumption.

Reducing power consumption doesn't just mean “green”, like recycle, save the polar bears.

The primary goal is to keep power consumption, and thus heat dissipation, below levels that will damage the CPU.

Over the last decade performance, especially on the server, has been dictated by adding more CPU cores.

It's no longer possible to make a single core run twice as fast, but if you add another cores you can provide twice the processing capacity — if the software can support it.

Modern processors are limited by memory latency not memory capacity

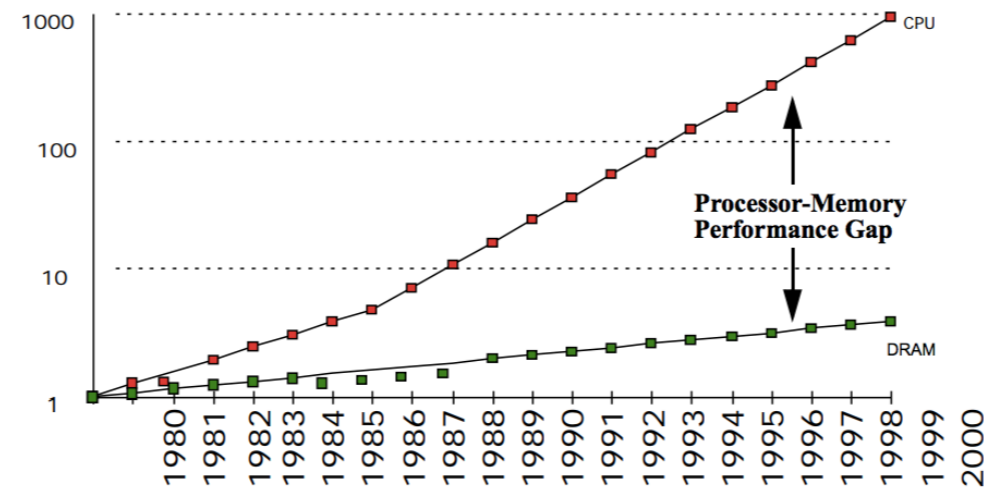


FIGURE 1. Processor-Memory Performance Gap.[Hen96].

Meanwhile, physical memory attached to a server has increased geometrically.

My first computer in the 1980's had kilobytes of memory.

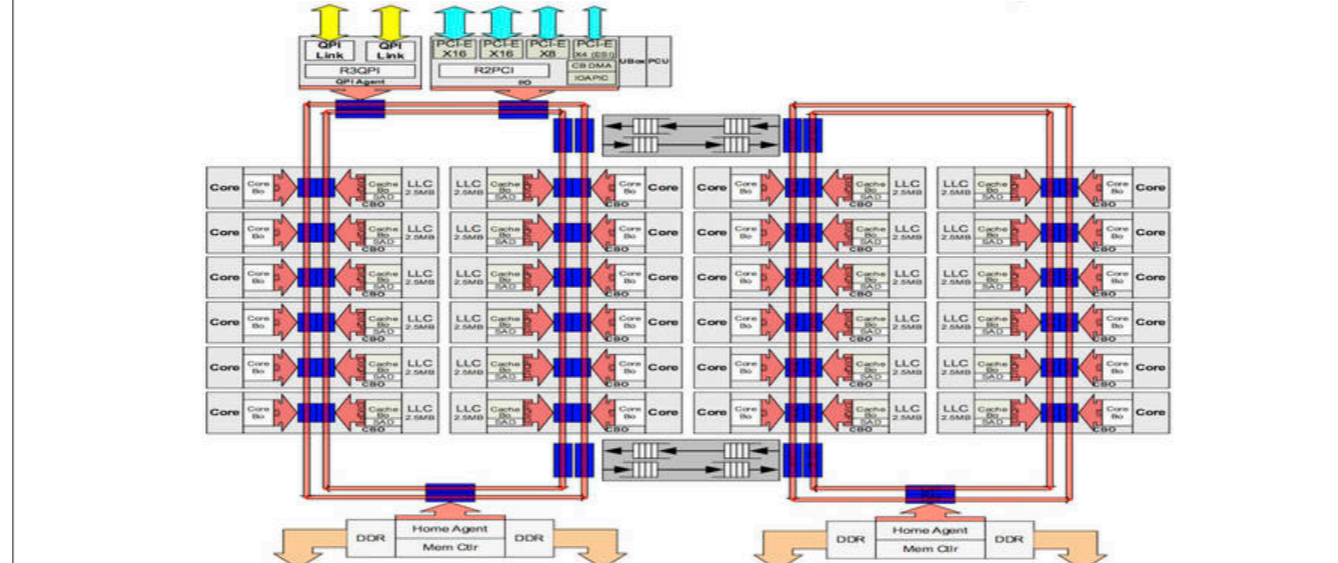
When I went through high school I wrote all my essays on a 386 with 1.8 megabytes of ram.

Now its commonplace to find servers with tens or hundreds of gigabytes of ram, and the cloud providers are pushing into the terabytes of ram.

But, in terms of processor cycles lost waiting for memory, physical memory is still as far away as ever because memory has not kept pace with the increases in CPU speed.

So, most modern processors are limited by memory latency not capacity

Cache size is limited by the speed of light



And this means computers, and thus programmers rely heavily on caches to hide the costs of memory access.

But cache is small, and will remain small because the speed of light determines how large a cache can be for a certain latency.

You can have a larger cache, but it will be slower because, because in a universe where electricity travels a foot every nanosecond, distance equals latency.

We can see the effects of this on this diagram of a Nehalem Xeon cpu. Each core has a little piece of the cache, and depending on which cache the information is stored in the cache lookup might have to transition several rings.

What does this have to do with programming languages?

So, why am I rambling on about hardware at a software conference ?

There's an old adage, that a slow programming language doesn't matter because it was cheaper to buy faster hardware next year than invest that time in optimising the program.

A decade ago, if you were working on a big python, or ruby, or perl application, buy the time you went to buy your next server, the same money you'd spent last year would buy you twice the performance.

This isn't true today.

“The free lunch is over”

-Herb Sutter (2005)

In 2005 Herb Sutter, the C++ committee leader, wrote an article entitled *The free lunch is over*.

In his article Sutter discussed all the points I covered and asserted that programmers could no longer rely on faster hardware to fix slow programs—or slow programming languages.

Now, a decade later, there is no doubt that Herb Sutter was right. Memory is slow, caches are too small, CPU clock speeds are going backwards, and the simple world of a single threaded CPU is long gone.

Moore's Law is still in effect, but for all of us in this room, the free lunch is over.



It's time for software to do its part

It's time for the software to come to the party.

Rick Hudson, one of the architects of Go's garbage collected noted at GopherCon in 2015, it's time for a programming language that works with the limitations of today's hardware, rather than continue to ignore the reality that CPU designers find themselves.

Hudson called it a "virtuous cycle". If we write programming languages that work well with the hardware, then hardware vendors will improve their hardware to support those patterns.

Modern hardware needs a programming languages which

- Are compiled, not interpreted.
- Permit efficient code to be written.
- Let programmers talk about memory effectively, think structs vs java objects.
- Can scale to multiple cores

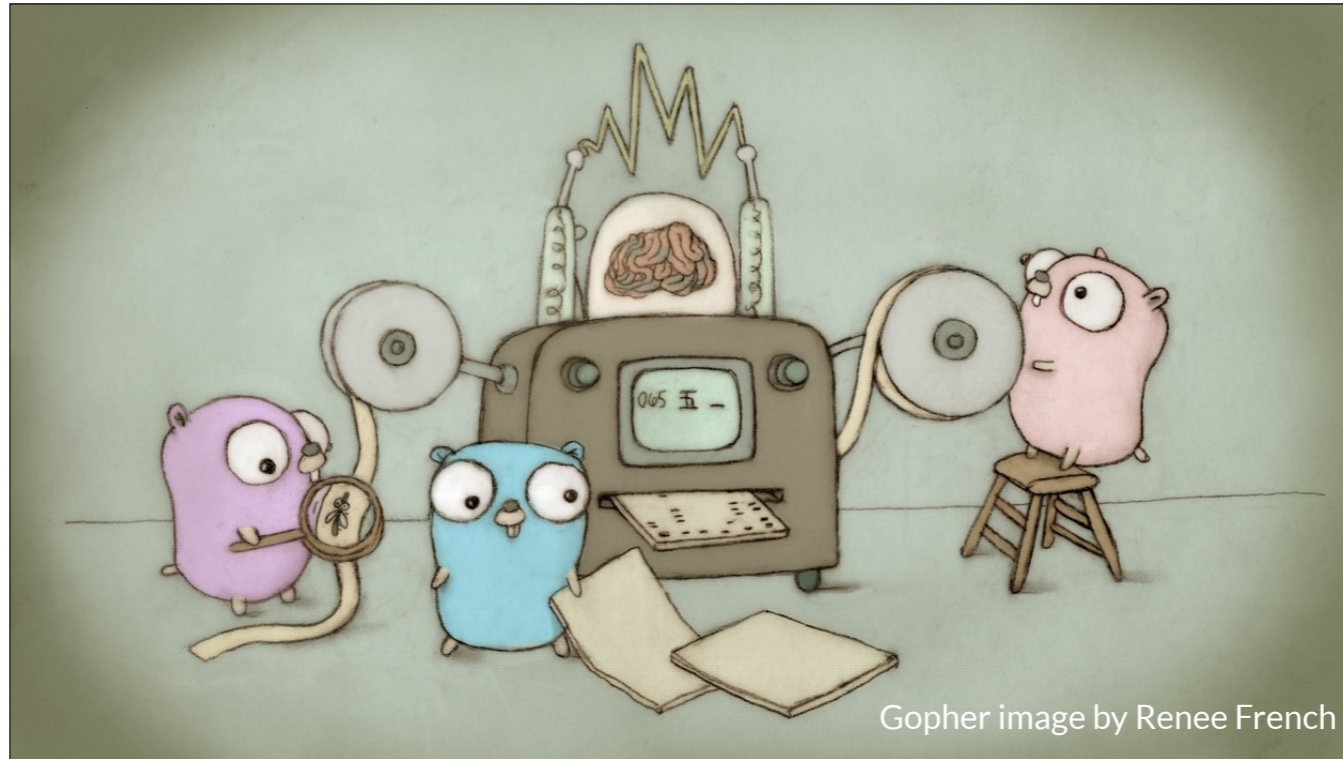
So, for best performance on today's hardware in today's world, you need a programming language which:

Is compiled, not interpreted, because interpreted programming languages operate poorly with CPU branch predictors and speculative execution.

You need a language which permits efficient code to be written, it needs to be able to talk about bits and bytes, and the length of an integer efficiently, rather than pretend every number is an ideal float.

You need a language which lets programmers talk about memory effectively, think structs vs java objects, because all that pointer chasing puts pressure on the CPU cache and cache misses burn hundreds of cycles.

Can scale to multiple cores



Obviously the language I'm talking about is the one we're here to discuss: Go.

Go, on the server

- Static binaries
- Powerful concurrency
- Efficient compiled code.

A common refrain when talking about Go is it's a language that works well on the server; static binaries, powerful concurrency, and good performance.

This talk focuses on the last two items, how the language and the runtime transparently let Go programmers write highly scalable network servers, without having to worry about thread management or blocking I/O.

Goroutines, the foundation of Go's concurrency story

Go has goroutines which are the foundation for its concurrency story.

I want to step back for a moment and explore the history that leads us to goroutines.



In the beginning, computers ran one job at a time in a batch processing model.

Each programmer would have complete control of the machine until their job was complete (or it crashed)

Computer time was highly prized and programmers would spend many hours waiting for the results of their edit compile test loop. (sounds a little bit like being a C++ developer)



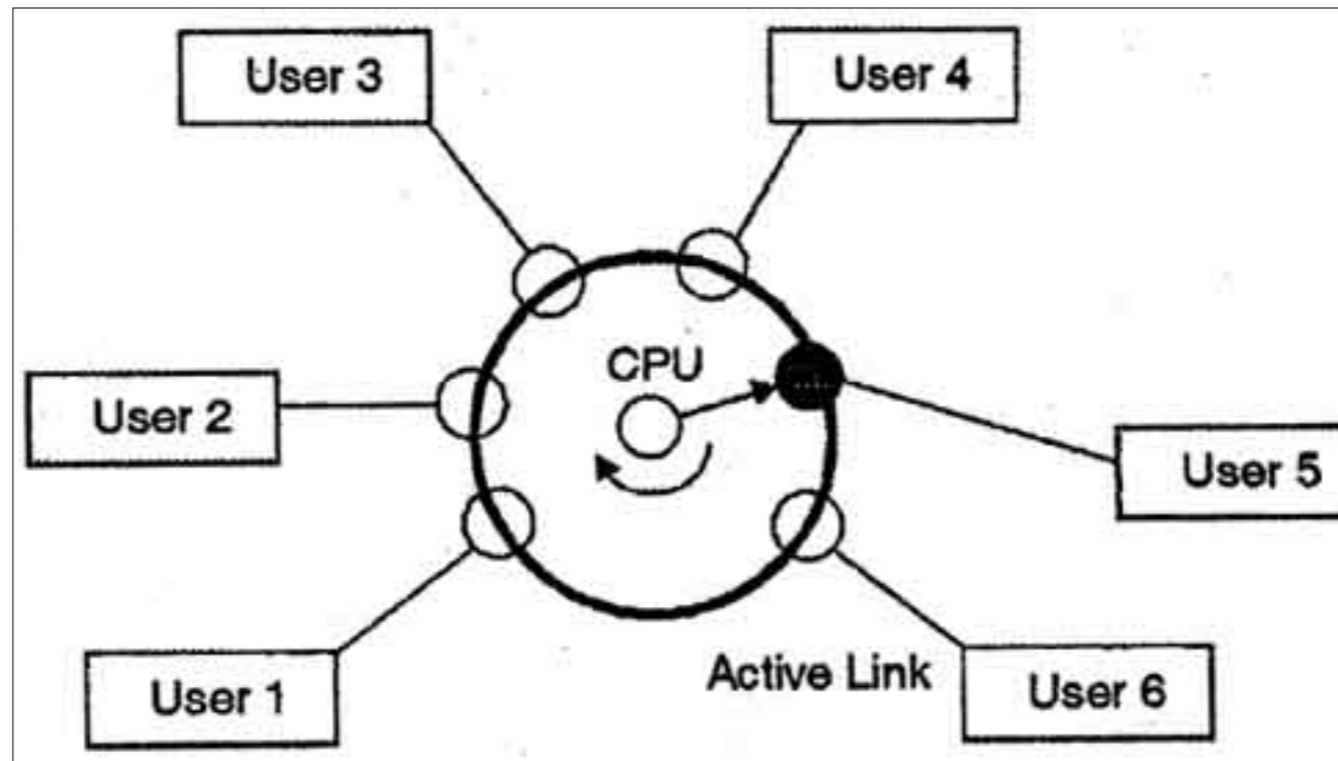
In the 1960's a desire for more interactive forms of computing lead to the development of time sharing, or multiprocessing, operating systems.



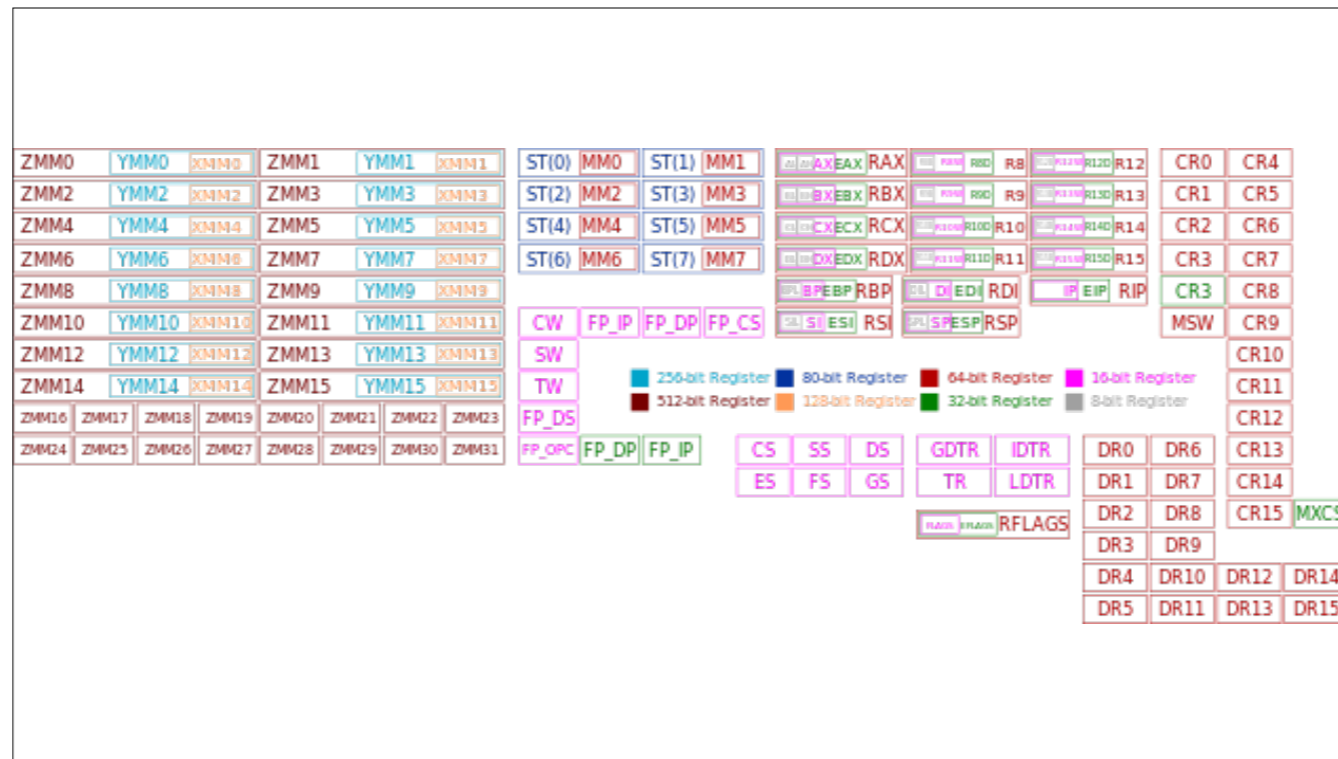
By the 70's this multi processing idea was well established for network servers like sendmail, ftp, telnet, rlogin,

This is Tim Burners-Lee's 25Mhz Next Cube, which in March of 1989 ran CERN httpd, the first web server.

CERN httpd handled each incoming network connections by forking a child process.



In a time-sharing system, the operating systems maintains the illusion of concurrency by rapidly switching the attention of the CPU between active processes by recording the state of the current process, then restoring the state of another. This is called context switching.

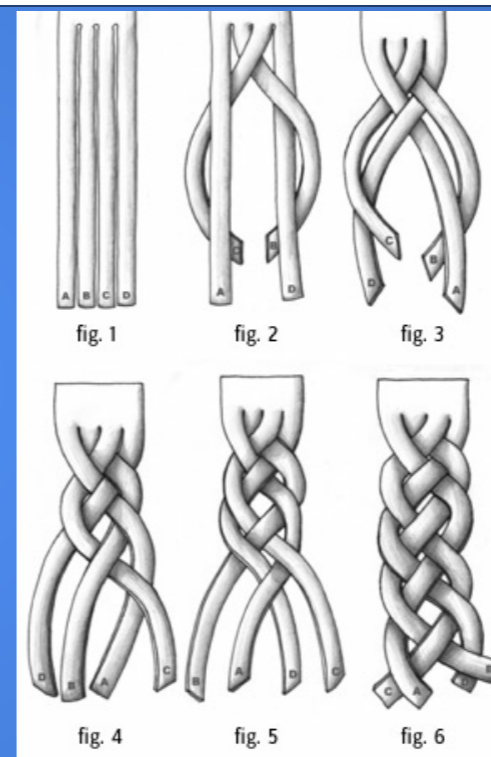


There are three main costs of a context switch.

1. The kernel needs to store the contents of all the CPU registers for that process, then restore the values for another process. Because a context switch can occur at any point in a process' execution, the operating system needs to store the contents of all of these registers because it does not know which are currently in use.
2. The kernel needs to flush the CPU's virtual address to physical address mappings (TLB cache)
3. Overhead of the operating system context switch, and the overhead of the scheduler function to choose the next process to occupy the CPU.

These costs are relatively fixed by the hardware, and depend on the amount of work done between context switches to amortise their cost—rapid context switching tends to overwhelm the amount of work done between context switches.

Threads



This led to the development of threads, which are conceptually the same as processes, but share the same memory space.

As threads share address space, they are lighter to schedule than processes, so are faster to create and faster to switch between. Multi-threaded processes can do more than one thing at a time if you have more than one physical CPU available.

However, threads are an abstraction created by the operating system so they incur the expensive context switch cost; a lot of state must be retained.

Goroutines

- Channel send and receive operations, if those operations would block.
- The Go statement, although there is no guarantee that new goroutine will be scheduled immediately.
- Blocking syscalls like file and network operations.
- After being stopped for a garbage collection cycle.

Goroutines take the idea of threads a step further.

Goroutines are cooperatively scheduled, rather than relying on the kernel to manage their time sharing.

The switch between goroutines only happens at well defined points, when an explicit call is made to the Go runtime scheduler.

- Channel send and receive operations, if those operations would block.
- The Go statement, although there is no guarantee that new goroutine will be scheduled immediately.
- Blocking syscalls like file and network operations.
- After being stopped for a garbage collection cycle.

In other words, places where the goroutine cannot continue until it has more data, or more space to put data.

Goroutines are multiplexed onto operating system threads

- Goroutines are cheap to create.
- Goroutines are cheap to switch between as it happens in user-space.
- Tens of thousands of goroutines in a single process are the norm; hundreds of thousands are not unexpected.

Many goroutines are multiplexed onto a single operating system thread.

- Super cheap to create.
- Super cheap to switch between as it all happens in user space.
- Tens of thousands of goroutines in a single process are the norm, hundreds of thousands not unexpected.

This results in relatively few operating system threads per Go process, with the Go runtime taking care of assigning a runnable Goroutine to a free operating system thread.

If you've had experience with Windows Fibres, stalkless python, the old user space threading model in Solaris or Linux, then you may be thinking “uh-oh” at this point.

Let me assure you that in practice this user space scheduler works well. This is because it is integrated with the runtime.

Compare this to threaded applications, where a thread can be preempted at any time, at any instruction. In Go, the compiler handles this as a natural byproduct of the function call preamble.

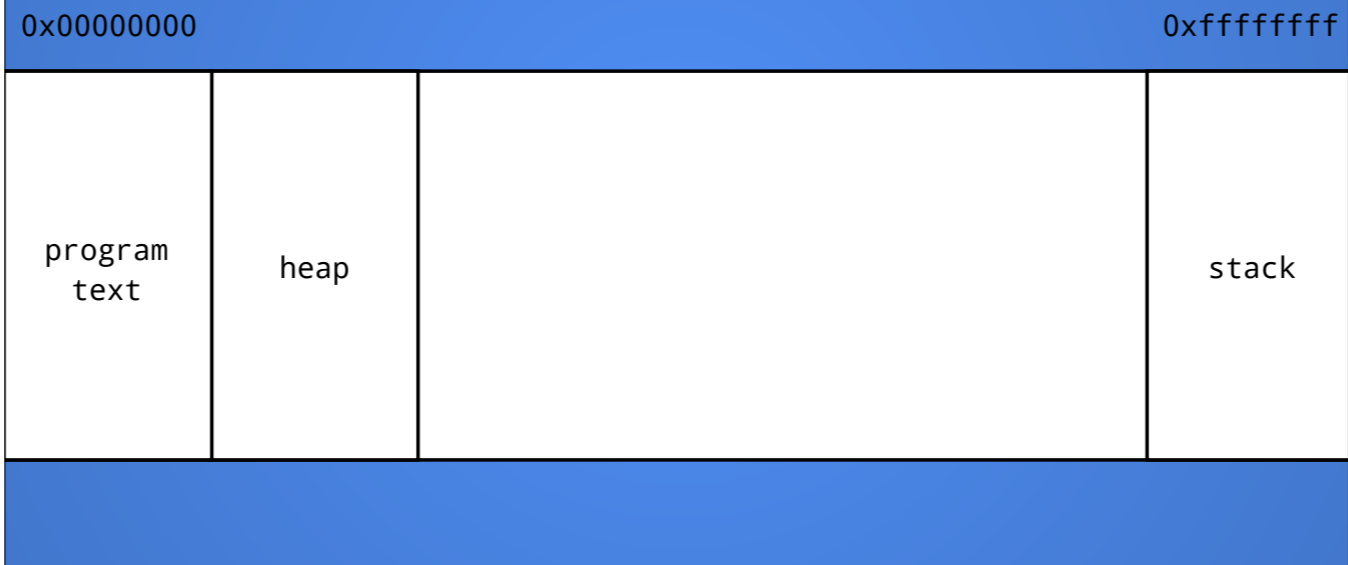
From the point of view of the language, scheduling looks like a function call, and has the same function call semantics. The thread of execution calls into the scheduler with a specific goroutine stack, but may return with a different goroutine stack.

Stack management

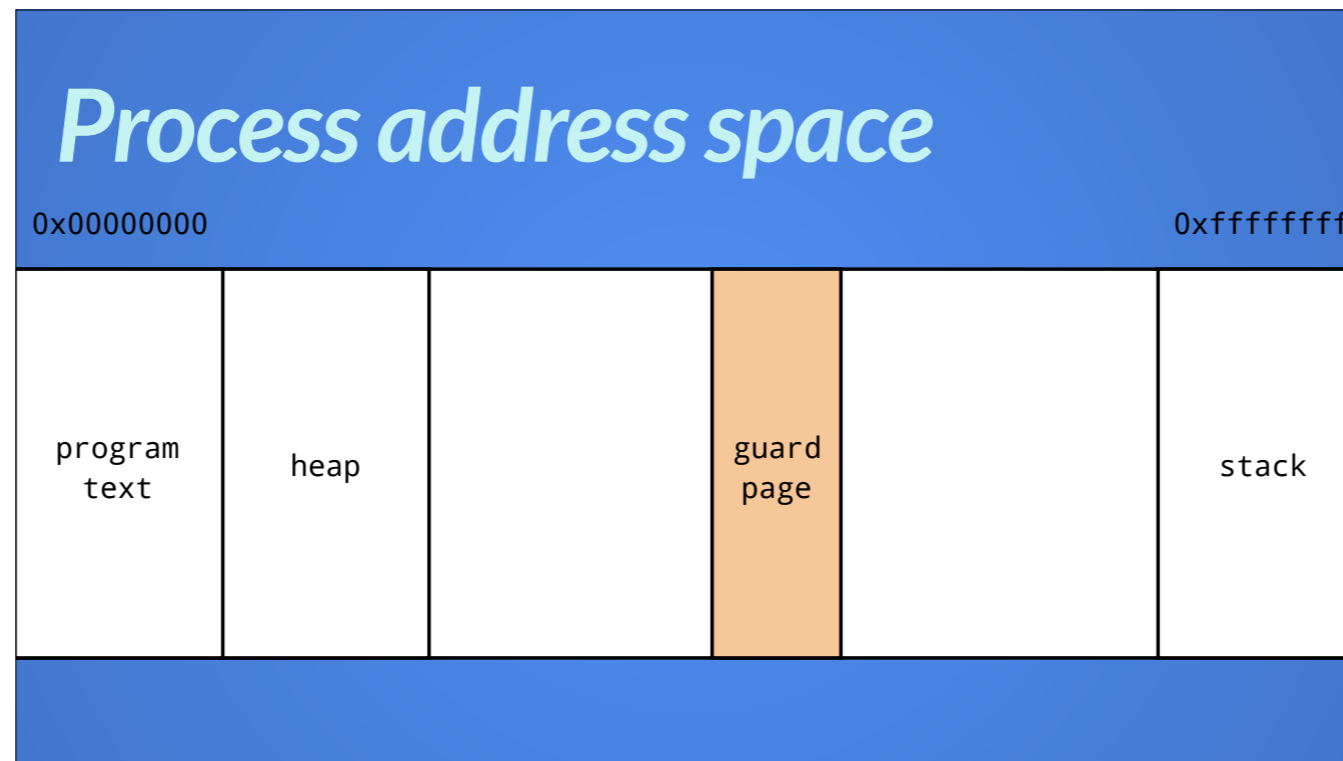
Goroutines allow programmers to create very cheap threading models, without the need for callbacks, or promises like you find in Twisted python or Node.js. This leads to an easy to read, imperative programming style.

There is another side to the goroutine story, and that is stack management.

Process address space



This is a diagram of the memory layout of a process. The key thing we are interested is the location of the heap and the stack.



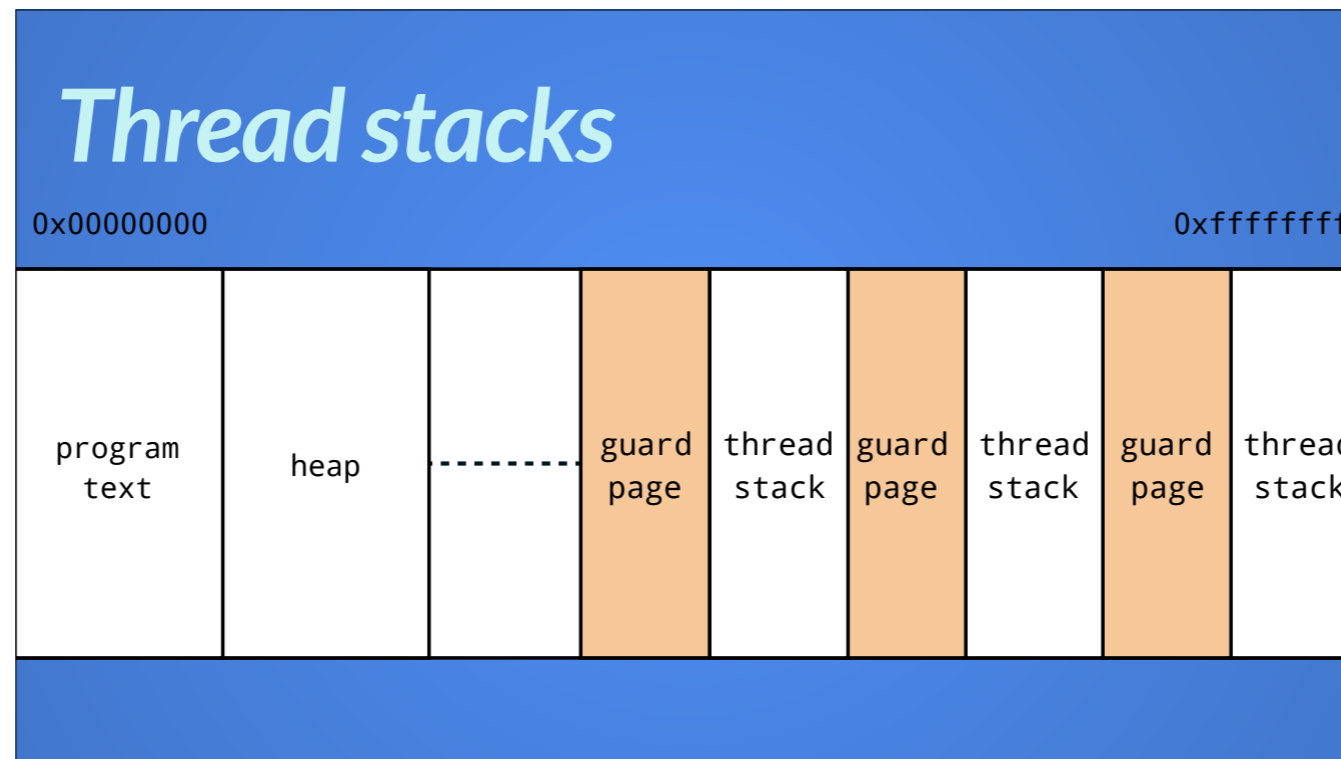
As the program runs, the heap is going to grow upwards, and the stack traditionally grows downwards.

(click)

Because the heap and stack overwriting each other would be catastrophic, the operating system arranges an area of unwritable memory between the stack and the heap.

(click)

This is traditionally called a “guard page”



Threads share the same address space, so for each thread, it must have its own stack, and its own guard page.

(click)

as we add more threads

(click)

Each comes with its own stack space sandwiched between two guard pages.

Because it is hard to predict the stack requirements of a particular thread, a large amount of memory must be reserved for each thread's stack. The hope is that this will be more than needed and the guard page will never be hit.

The downside is that as the number of threads in your program increases, the amount of available address space is reduced.

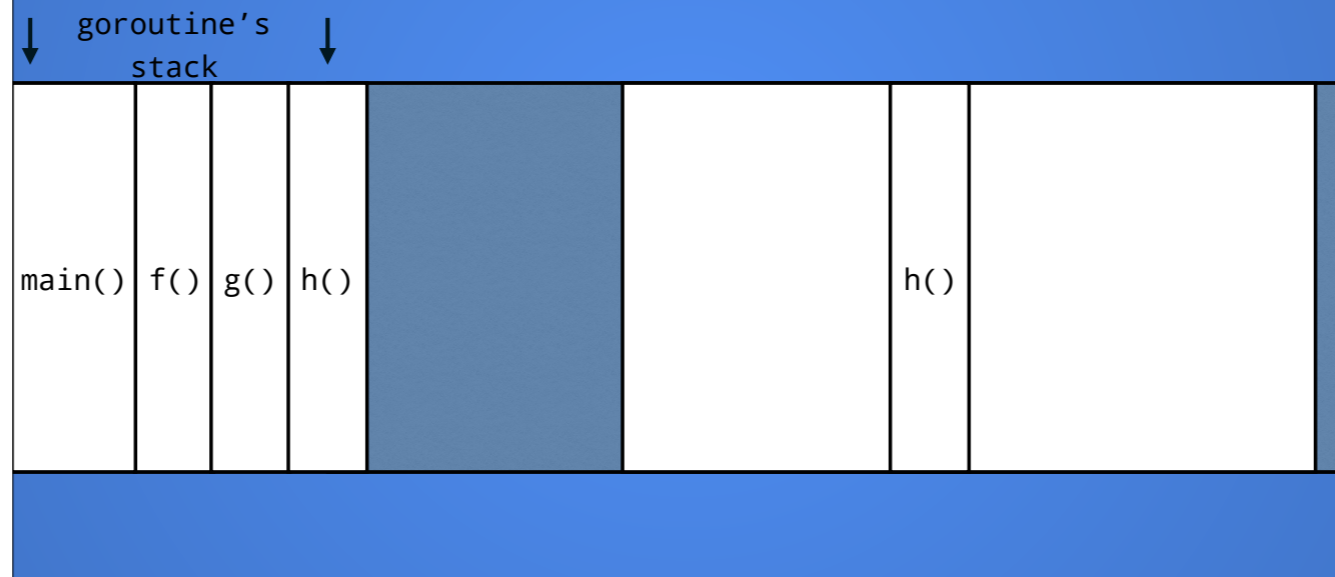
The early process model, allowed the programmer to view the heap and the stack as effectively infinite. The downside was a complicated and expensive subprocess fork/exec model.

Threads improved the situation a bit, but require the programmer to guess the most appropriate stack size; too small, your program will abort, too large, you run out of virtual address space.

Goroutine stack management

We've seen that the Go runtime schedules a large number of goroutines onto a small number of threads, but what about the stack requirements of those goroutines ?

Goroutine stack growth



Each goroutine starts with a small stack, about 2 kilobytes, allocated from the heap.

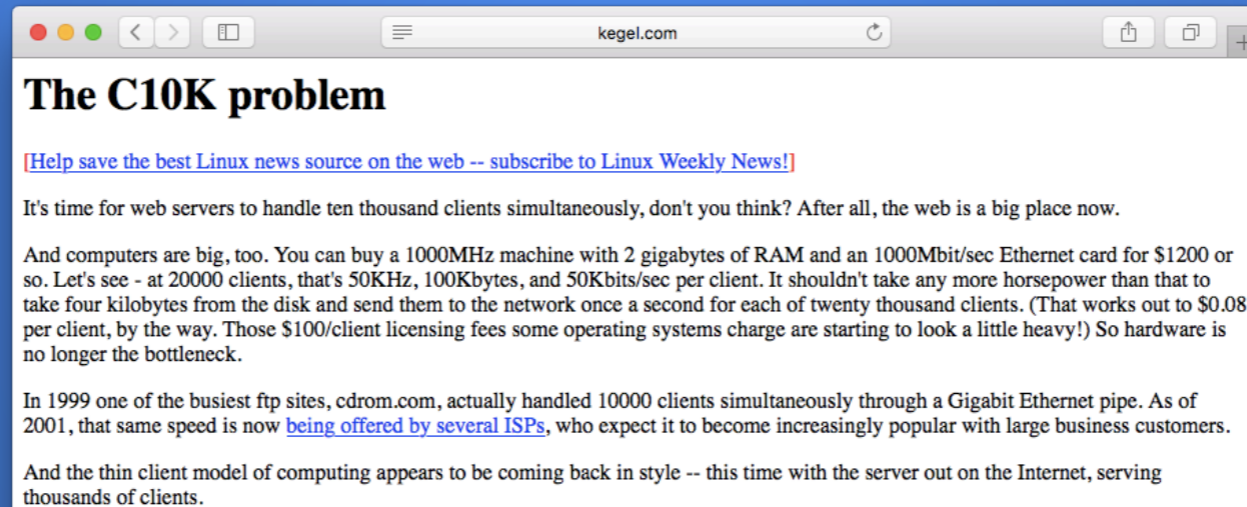
Instead of using guard pages, the Go compiler inserts a check as part of every function call to test if there is sufficient stack for the function to run. If there is sufficient stack space, the function runs as normal.

If there is insufficient space, this check will fail and trap into the Go runtime.

The runtime will allocate a larger stack segment on the heap, copy the contents of the current stack to the new segment, free the old segment, and the function call is restarted.

Because of this check, a goroutine's initial stack can be made much smaller, which in turn permits Go programmers to treat goroutines as cheap resources. Goroutine stacks can also shrink if a sufficient portion remains unused. This is handled during garbage collection.

Integrated network poller



The third feature of Go I want to talk to you today is the so called “integrated network poller”

(click)

In 2002 Dan Kegel published what he called the c10k problem.

Simply put, C10k asked the question of how to write a server that can handle at least 10,000 TCP sessions on the commodity hardware of the day. Since that paper was written, conventional wisdom has suggested that high performance servers require native threads, or more recently, event loops.

Event loops

- 👍 High scalability, efficiently handle lots of connections.
- 👎 Convoluted callback/errorback coding style.

Almost all operating systems support an efficient way to handle many network connections.

You take all your connections, pass them to an operation like kqueue, or select, or epoll, and then when one becomes ready you're notified.

This is the soul of an event loop.

Are there any node.js fans in the room?

What do you like about node? High scalability, lots of connections.

What don't you like about node? The callback nature of the coding style.

Threads carry a high overhead in terms of their scheduling cost and memory footprint.

Event loops ameliorate those costs, but introduce their own requirements for a complex, callback driven style.

Go provides programmers the best of both worlds.

Go's answer to C10k

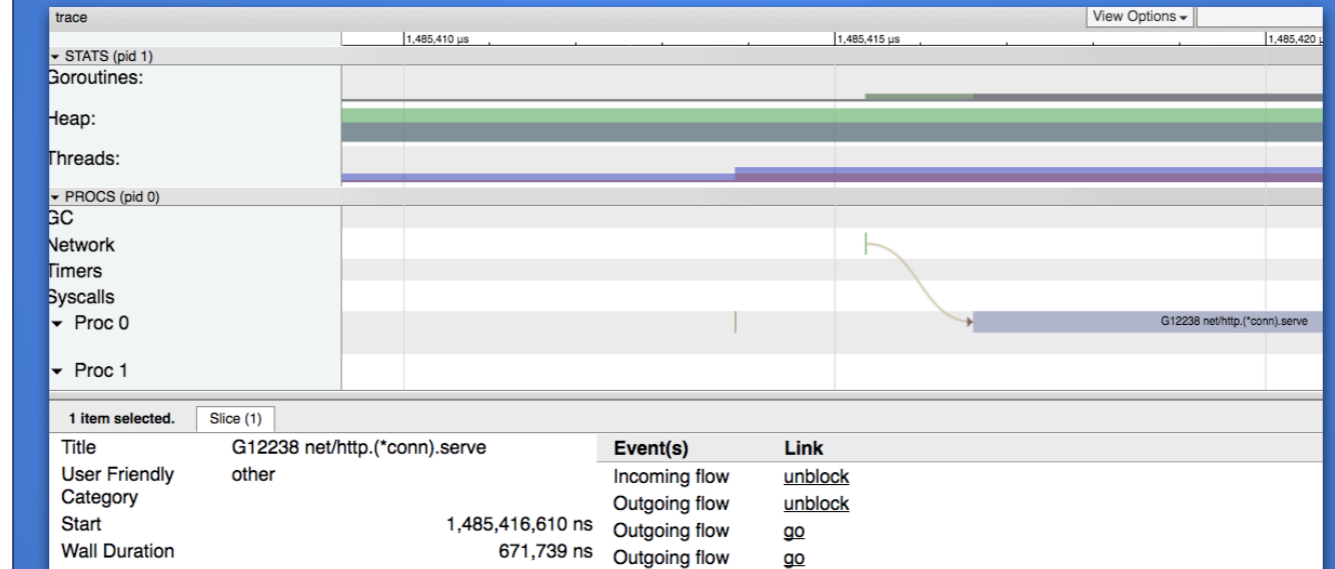
In Go, syscalls are usually blocking operations, this includes reading and writing to file descriptors. The Go scheduler handles this by finding a free thread or spawning another to continue to service goroutines while the original thread blocks.

However for network sockets, by design at any one time almost all of your goroutines are going to be blocked waiting for network IO. This is what highly concurrent servers do.

A naive implementation of this would have one goroutine calling into kqueue or epoll, and when it received readiness notifications from sockets, it would pass those over a channel to the original goroutine that issued a network read or write.

This achieved the goal of avoiding a thread per syscall overhead, by using the generalised wakeup mechanism of channel sends.

Go's scheduler is network aware



There's nothing inherently wrong with it, but because network transfers are highly latency sensitive, in a busy server there could be a relatively long delay between a readiness notification occurring and the original goroutine being woken up to service the notification.

In Go, the network poller has been integrated into the runtime itself. As the runtime knows which goroutine is waiting for the socket to become ready the scheduler can put the goroutine back on the same CPU as soon as the packet arrives.

This has the effect of reducing IO latency; draining os buffers quickly and keeping tcp windows open and transmission rates high.

The image here is the trace viewer, which is the tool I'll be talking about tomorrow at my workshop,

Goroutines, stack management, and an integrated network poller

- Goroutines provide a powerful abstraction that frees the programmer from worrying about thread pools or event loops.
- The stack of a goroutine is as big as it needs to be without being concerned about sizing thread stacks or thread pools.
- The integrated network poller lets Go programmers avoid convoluted callback styles while still leveraging the most efficient IO completion logic available with the operating system.

In summary

Goroutines provide a powerful abstraction that can free you, as the programmer, from worrying about thread pools or event loops.

The stack of a goroutine is as big as it needs to be without being concerned about over allocating thread stacks or thread pools.

The integrated network poller lets Go programmers avoid convoluted callback styles while still leveraging the most efficient IO completion logic available with the operating system.

The runtime makes sure that there will be just enough threads to service all your goroutines and keep your cores active.

And the thing that I want you all to take away, and the reason that I came here to Russia to explain this to you is, all of these features are transparent to the programmer.

You just write normal imperative Go code, and you get all of these features for free.

But wait, there's more


- Super fast compilation.
- Excellent tooling; go {build, test, fmt, vet, lint, doc}, built in race detector.
- Excellent cross platform support; cross compilation without tears.
- Single static binary enables a simple deployment story.

As this is the performance talk, I've mostly constrained my remarks to this area, but there is a lot more to the Go success story than just being fast, and these are arguably the more notable features of the language.

- Super fast compilation.
- Excellent tooling; go build, go test, go fmt, go lint, go vet, godoc, -race detector.
- Excellent cross platform support, cross compilation without tears
- Single static binary enables a ridiculously simple deployment story, scp(1) and you're done -- no runtime interpreter required.

Would you like to know more?

Go take the tour  tour.golang.org

Go join a meetup  go-meetups.appspot.com

Go watch some talks  gophervids.appspot.com



Thank you!

@davecheney · dfc@heptio.com

<https://dave.cheney.net>

Thank you again.