University of Salzburg
Department of Computer Science

Google Go!

A look behind the scenes

Seminar for Computer Science
Summer 2010

Martin Aigner
Alexander Baumgartner

July 15, 2010

Contents

1 Introduction

Go is a programming language with a focus on systems programming, i.e. writing code for servers, databases, system libraries, browsers and so on. It was created because the authors of Go were frustrated with the existing languages for systems programming and the tradeoff one had to make between execution time, compilation time and the time for development. Dynamic languages like Python allow fast development but suffer from runtime overhead. Static languages like C allow fast execution but suffer from flexibility when writing large scale programs. C++ was designed to add some of this missing features to C but this leads to complex compilers, long compilation time and huge object files with a lot of duplicated code which requires smart linkers to keep the code footprint small.

Go is designed to meet the needs of system programmers combining the useful concepts of different worlds. It is a compiled language with a static type system i.e. strong type checking is performed at compile time which avoids a lot of errors that programmers tend to make in dynamic typed languages. Dynamic memory allocation is done with garbage collection. Thereby the complexity of correct memory deallocation drops. It gets hard to write unsafe code in Go!

Go is object-oriented but unlike other object-oriented languages like Java or C++ Go is not type-oriented. There is no subclassing because there are no classes at all. Instead methods can be defined on any type such as basic types like int, float or bool but also on composite types like maps or structs. Hence, objects in Go are simply types with methods defined on them.

The behavior or abilities of objects in Go can be specified using *interfaces*. Interfaces are a set of method declarations and if this set is contained in the set of methods on a given type then this type satisfies implicitly the interface. Note that an object may satisfy multiple interfaces. In Go interfaces represent abstraction and structs (or even simpler types) represent data. Due to this orthogonality it becomes very easy to change Go programs to new requirements.

Concurrency is important in systems programming especially for reactive systems and parallel computation. The main challenge in concurrent programming is to get the synchronization on shared resources right. Go is intended to aid the programmer with concepts and idioms that make concurrent programming easier. Parallelism is not the primary goal behind the concurrent concept. Nevertheless if a program is structured

in a way to perform multiple tasks at the same time then it can be run in parallel thus achieving a significant speedup. Still the focus is on writing code that is easy to understand and easy to change without breaking the whole concurrency concept of a program.

In this work we focus on the representation of data of the Go compiler and the Go runtime system so one can get an intuition about which operations are expensive and which are not. We will also discuss the concepts that allow elegant concurrent programming and the runtime support that enables thousands of concurrent routines to be executed efficiently.

2 Data representation in Go

Good systems programmers know what the compiler or interpreter does with the code they write and how it affects the execution of the program. A good starting point to understand how a Go program is executed is to understand the data structures Go uses to represent data.[1]

2.1 Basic types and arrays

The basic types in Go are similar to other languages and most programmers should be familiar with them. There are types like `int` or `float` which represent values of a size that might depend on the target machine (e.g. 32 or 64-bit). But there are also types with explicit size like `int8` or `float64`. Note that on a 32-bit machine `int` and `int32` may have the same size but nevertheless are distinct types and assigning a value of type `int32` to a variable of type `int` will cause a compile time error.

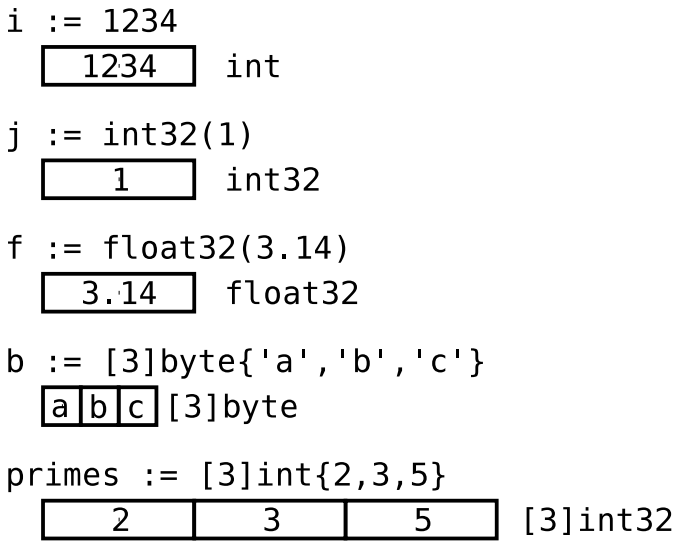


Figure 2.1: Memory Layout of basic types

The examples used here show a 32-bit memory layout. Note that in the current imple-

mentation of Go on a 64-bit machine only pointers are 64 bits long while `int` still uses 32 bits.

Figure 2.1 shows the memory layout of some basic types. The variable `i` has the implicit 32-bit type `int` while the variable `j` has the explicit 32-bit type. As mentioned above an assignment `i = j` would fail and must be written with an explicit conversion `i = int(j)`.

The variable `f` has the same memory footprint as `i` or `j` but a different interpretation.

Arrays in Go are similar to arrays in C. The array `b` consists of 3 bytes of contiguous memory where type `byte` is a synonym for `uint8` and is used as the element type of strings. Similarly `primes` is an array of `int` with length 3 which consumes 3 * 4 = 12 bytes of memory.

2.2 Structs and pointers

Go lets the programmer decide what is a pointer and what is not. In Figure 2.2, a struct named Point is defined which is represented as two adjacent words (of type int) in memory.

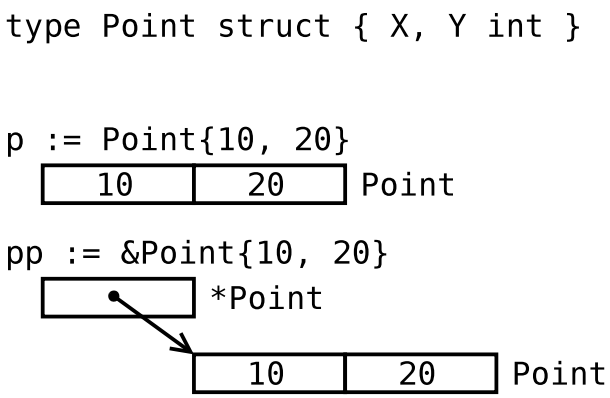


Figure 2.2: Memory Layout of structs

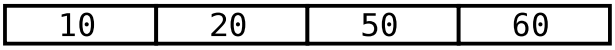
The composite literal syntax `Point{10, 20}` creates the initialized Point `p`. Like in C, the address operator `&` takes the address of a newly created and initialized `Point` in memory where `pp` points to. Composite literals are often used in Go to construct values for structs, arrays, slices, and maps. Given the composite literal, the compiler automatically determines the types of `p` and `pp`. The type of `p` is `Point` whereas the type of `pp` is `*Point`.

As we saw, the fields in a struct are laid out contiguously in memory. This also applies for composite types e.g. structs of structs. Figure 2.3 shows the memory layout of

variables of the type `Rect1` and `Rect2`. `Rect1` is a structure that contains two fields of type `Point` whereas `Rect2` consists of two pointers to values of type `Point`.

```
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }

r1 := Rect1{Point{10, 20}, Point{50, 60}}
```




```
r2 := Rect2{&Point{10, 20}, &Point{50, 60}}
```

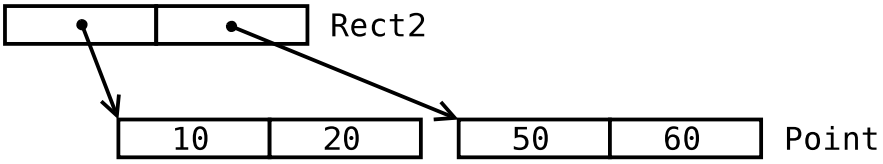


Figure 2.3: Memory Layout of composite structs

Programmers who are familiar with C will not be surprised about the distinction between values and pointers. It gives the programmer the freedom do control the memory layout of the data structures she builds. Often this is important for building well performing systems. Consider data structures that have to fit in one cache line for example.

2.3 Strings and slices

Go's strings are implemented in a different way than in C. In C, a string is the same as an array of `char` which is terminated with `'\0'`. In the following examples the gray arrows denote pointers that exist in the implementation of Go but are not visible to the program. Neither can they be read nor altered.

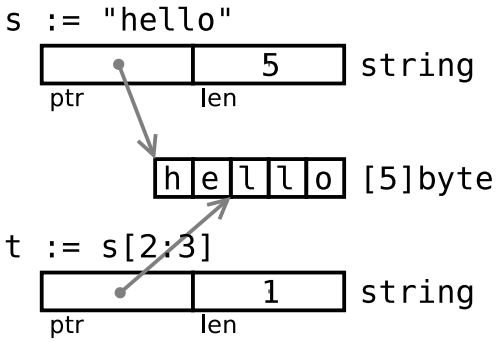


Figure 2.4: Memory Layout of a string

In Go, strings are represented as a two-word structure in memory. The first word is a pointer to the string data and the second word contains the length. Figure 2.4 depicts the declaration of a variable `s`. The Go compiler allocates the 5 bytes for the data and creates the two-word structure representing `s`.

Go defines strings to be immutable. So it is save for multiple strings to share the same underlying data. The variable `t` is of type `string` and is created using the *slice operation* on `s`. The slice operation is a primary expression on a string, array or slice which creates a substring or slice on the underlying data. If the sliced operand is a string or slice, the result of the slice operation is a string or slice of the same type. If the sliced operand is an array, the result of the slice operation is a slice with the same element type as the array [2]. The syntax of the slice operation is `a[lo:hi]` where the index expressions `lo` and `hi` select which elements appear in the result. The result has indexes starting at 0 and length equal to `hi-lo`. The example in Figure 2.4 leads to a string `t` that contains the single character `"l"`.

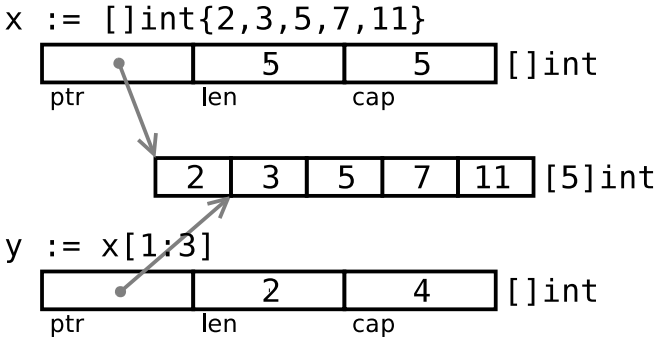


Figure 2.5: Slicing an array of integers

Now that we discussed the slice operation on strings we will take a look at the slice type. As shown in Figure 2.5, a slice consists of a tree-word data structure containing the pointer to the underlying data, the length and - in contrast to strings - the capacity of the slice. The length is the upper bound for indexing operations like `x[i]` while the capacity is the upper bound for slice operations like `x[i:j]`. Note the difference of the composite literal creating `x` and the one creating `primes` in Figure 2.1. The only difference is in omitting the size of the array which is determined automatically.

The advantage of slicing is that creating a new slice of a string or an array does not create a copy of the underlying data. It does not even invoke the allocator because most of the time the three-word slice will reside on the stack. This makes passing a slice as a function parameter as efficient as passing a pointer and length pair in C. However, like in Java there is a well-known drawback when keeping a small slice of a huge array. The existence of the slice keeps the entire data in live memory.

2.4 Dynamic allocation with “new” and “make”

Data structures in Go can be created with two different functions: `new` and `make`. The distinction may seem confusing but programmers who are new to Go will get used to it quickly. The basic difference is that `new(T)` returns `*T`, a pointer to a variable of type `T`, while `make(T, args)` returns an ordinary `T`.

Figure 2.6 depicts the memory representation that is generated by using `new`. Note that `new` creates data structures with zeroed memory.

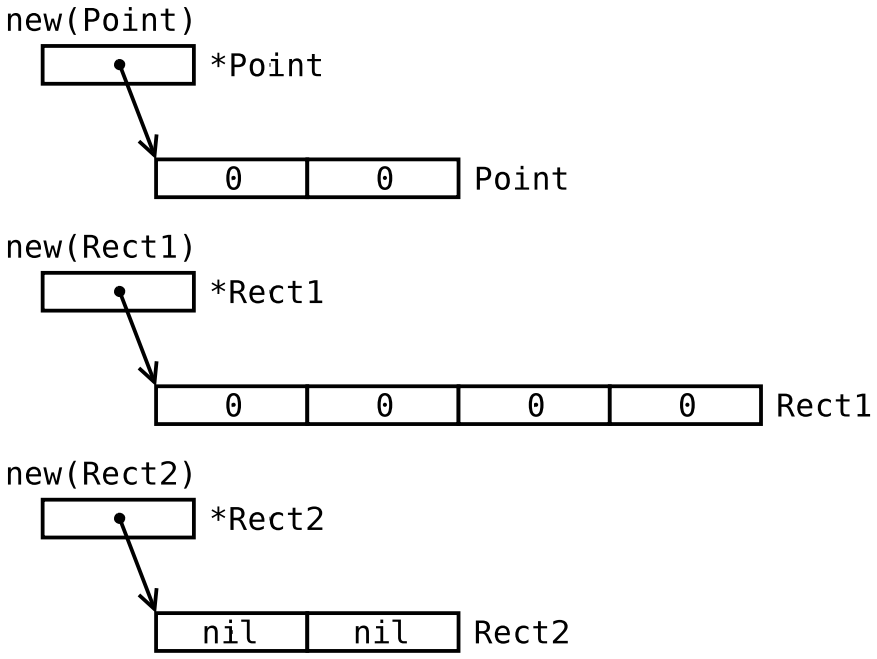


Figure 2.6: Allocation with “new”

Figure 2.7 shows the difference of `make` and `new` on a slice of int. Again, the statement `new([]int)` creates a pointer to an uninitialized slice. This is the equivalent to zeroed memory. On the other hand `make([]int, 0)` initializes the slice with the underlying data structure, even if it is empty. The example with `make([]int, 2, 5)` depicts a typical use of `make`. Again the underlying data structure is initialized and ready to use. The rule is, that `make` is used for channels, slices and maps only because these are complex data structures and require such initialization. If `new`-allocated types need initialization then one can provide an initializing constructor.

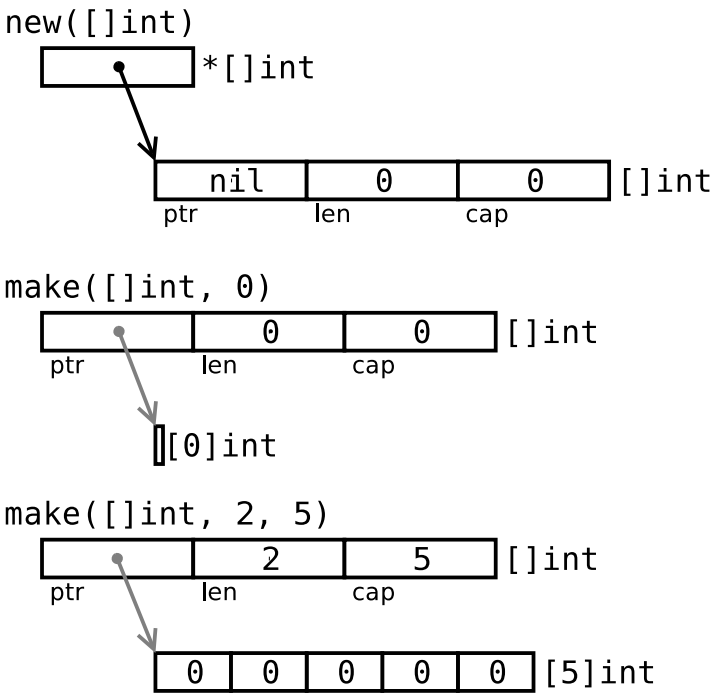


Figure 2.7: Allocation with "make"

2.5 Maps

In Go, maps are built-in data structures. There are no extra imports needed like in Java. Go maps are provided through the runtime package which is linked to every Go program by default.

Listing 2.1: Using maps in Go

```
1  //composite literal construction
2  var timeZone = map[string] int {
3      "UTC":  0*60*60,
4      "EST": -5*60*60,
5      // and so on
6  }
7  //accessing map values
8  offset := timeZone["EST"]
9
10 //checking 0 v.s. non-existence
11 var seconds int
12 var ok bool
13 seconds, ok = timeZone[tz] //comma ok idiom
```

The design decision for built-in maps was made because of the benefit of syntactical support for maps in the language. In Java, maps are used with the correlative methods

whereas Go supports map usage with built-in idioms similar to accessing elements of an array. In the current Go release, maps are implemented as hash maps.

Listing 2.1 shows a simple map handling example in Go. Like arrays or strings, maps can be created with a composite literal (Line 2-6). Thereby the map data structure is created under the hood. The key can be of any type for which the equality operator is defined. In this example, the type of the key is `string`. Structs, arrays and slices cannot be used as map-keys, because equality is not defined on those types. Like slices, maps are a reference type. If you pass a map to a function that changes the contents of the map, the changes will be visible in the caller.[2] Line 8 is an example for the syntactic support of map access in Go.

An attempt to fetch a map value with a key that is not present in the map will return the zero value for the type of the entries in the map. For instance, if the map contains integers, looking up a non-existent key will return 0. If the distinction between a missing item from the zero value is necessary then the *comma ok* idiom may be used (Line 13). The map access then returns both the (possible zero) value and a boolean flag that is true only if an entry for `tz` exists in `timeZone`.

2.6 Implementation of interface values

Interfaces are one of the most interesting things in the Go language. They are used to abstract and describe the behavior of objects. Interfaces enable the use of *duck typing*[1] like one would would expect from dynamic languages like Python, but still let the compiler catch some typical mistakes programmers tend to make.

Let us consider the following example: Listing 2.2 defines an interface called `Stringer`. The only method defined is `String()` which returns a `string`. Furthermore we define a type `Binary` to be a 64-bit integer and two methods on this type. The first method is a `Get()` function that simply returns the value of the `Binary` converted to an integer. The second method is called `String()` and returns a bit-string, i.e. a string representation of the binary value to the base of two.

Note that we do not have to tell `Binary` to implement `Stringer`. The runtime can see that `Binary` has a `String()` method, so it implements `Stringer`. Consider the case where the source code defining `Binary` is not available and the `Stringer` interface is added post-facto. This would not be possible e.g. in Java because interface implementation has to be explicit there. The implicit implementation of interfaces in Go is one of the strongest features for refactoring. The abstraction can be created while the code is written without ending in a mess.

[1] Duck typing is a style of dynamic typing in which an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface [3].

Listing 2.2: Using interfaces in Go

```
1  type Stringer interface {
2      String() string;
3  }
4
5  type Binary uint64
6
7  func (i Binary) Get() uint64 {
8      return uint64(i)
9  }
10
11 func (i Binary) String() string {
12     return strconv.Uitob64(i.Get(), 2)
13 }
```

Languages with methods, i.e. languages that support dynamic binding typically fall into one of two categories: statically prepare method-call tables, which occurs in C++ and Java, or do a method lookup for each call like the Smalltalk family of languages does. Go is somewhere in between. It has method tables like C++ but these are computed at runtime.

Let us stick to the previous example. A `Binary` (Figure 2.8) consists of two words of memory (again on a 32-bit architecture).

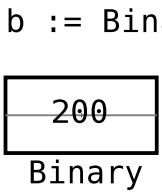


Figure 2.8: A Binary is a 64-bit integer. Note that memory grows down and not to the right as in previous examples

Interfaces values are represented as a two-word pair giving a pointer to information about the type and a pointer to the underlying data. Assigning `b` to a value of type `Stringer` results in the data structure in Figure 2.9.

The assignment from `b` to `s` makes a copy of the data stored in `b`. This is as straight forward as with any other assignment. If `b` changes later, `s` is not affected at all.

The first word in the interface value points to the so called interface table (itable). This table begins with some metadata about the types involved followed by a list of function pointers. The interface table corresponds to the interface type and not the dynamic type. In our example the itable for `Stringer` holding type `Binary` lists the methods used to satisfy `Stringer`. The other methods that the dynamic type may have (`Get()`

in our example) do not appear in itable. The second word in the interface value is a pointer to the actual data. As mentioned above, this is a copy of `b`.

To check if an interface value holds a particular type, the Go compiler generates code like (C-equivalent) `s.tab->type` to obtain the type pointer and check it against the desired type. If the types match, then `s.data` is valid to use.

To call `s.String()` the compiler generates the following code (C-equivalent): `s.tab->fun[0](s.data)`, i.e. it calls the right function pointer from the interface table.

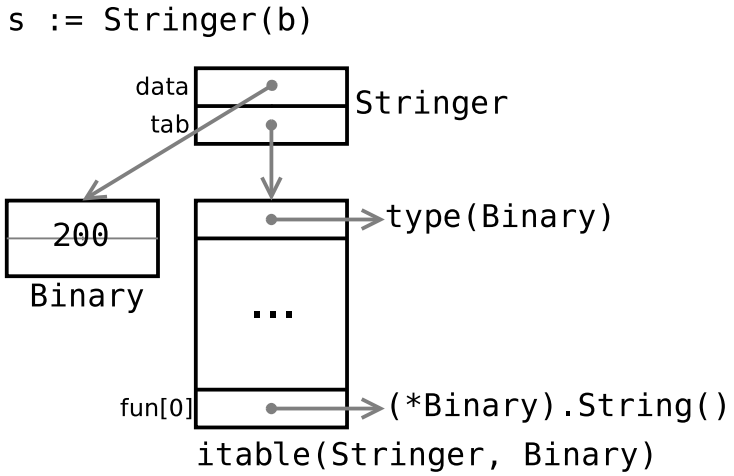


Figure 2.9: Interface representation: The gray pointers again are not visible to the Go program but to the runtime system.

The interface table is computed at runtime because there could be many pairs of interface types and concrete types and most of them would never be used. Instead the compiler generates a type description structure for each concrete type (like `Binary`) containing a list of methods that are implemented by that type. For interface types the compiler creates a similar list containing all methods that are defined in the interface. The runtime system fills the itable by looking for each interface method in the corresponding (runtime evaluated) concrete type's method table. The runtime caches the result i.e. it has to be computed only once per call. Thereby dynamic method calls can be done in constant time by dereferencing the appropriate function pointers.

3 The Go Runtime System

3.1 Library dependencies

An installation of Go comes with a standard library and a runtime system that provides a lot of features that already enable large scale applications. Still the amount of libraries that come with the base distribution is small compared to e.g. a Java runtime implementation. Go libraries are organized in packages that provide some sort of namespace like in C++ or C#. The interesting thing about Go packages in contrast to the separate compilation paradigm in C/C++ is that the dependency tree is not transitive, i.e. in C `#include <stdio.h>` reads hundreds of lines from 9 files because stdio.h has `#include` directives to other header files and so on. As libraries grow, the amount of library declarations to read grows exponential e.g. in Objective-C, `#include <Cocoa/Cocoa.h>` reads more than a hundred thousand lines from nearly 700 files without having any code generated yet![4]

In Go the dependency tree is truncated to the first package file that is used. For example `import "fmt"` reads one file where 195 lines summarize 6 dependent packages. This yields a huge improvement of compilation time. An improvement that becomes exponential as a Go program scales.

As mentioned above, Go ships with a significant yet manageable amount of standard libraries for example formatted printing, RPC, basic cryptography or XML processing. With the distribution also comes a tool called *goinstall* that is comparable to a distributed CPAN like in Perl or RubyGems where programmers can share or distribute their libraries and the installation becomes very easy.

3.2 Memory safety by design

The usage of dynamic memory in C/C++ is known to be error prone. The programmer allocates a peace of heap memory with library functions like `malloc` or `new` and returns it to the memory manager with `free` or `delete`. Symptoms of errors in explicit memory management are memory leaks and program crashes due to segmentation faults. Both come from misplaced or missing `free` calls. Concurrent programming in combination with dynamic memory is even more difficult. Methods for proofing correctness of such

concurrent programs are complex and expensive and thus done seldom. A source of error is to track the ownership of objects as they move around threads. Not only the access of shared objects but also freeing them needs concurrent reasoning.

Garbage collectors solve the problem of correctly deallocating memory. Both tracing and reference counting garbage collectors compute reachability of objects either directly or indirectly [5]. An object is deallocated only if there are no more references pointing to it. Note that reachable memory leaks are still possible.

In systems programming garbage collectors are rarely used because of the runtime overhead and the response latency they introduce. The Go designers traded the reduced programming complexity against the costs of garbage collectors. Recent advances in garbage collection have the potential to make them usable for systems programming.[6][7]

Some other intrinsic safety additions are:

- Go has pointers but no pointer arithmetic. This enables call-by-reference functions without the error prone possibility to modify the pointer itself.
- Local variables move to the heap as needed. This is important for supporting first-class functions and *goroutines* (as threads are called in go) because the programmer might pass a reference to a local variable to a goroutine and then return from the current stack frame thus invalidating stack-allocated data. Heap-allocated locals are deallocated by the garbage collector.
- As mentioned in section 2.3 all indexing is bounds-checked.
- All variables are zero-initialized.
- There is no pointer-to-integer conversion. However, package *unsafe* enables this feature but labels the code as dangerous. This can be used for some low-level libraries but should not be used in mission critical programs.

3.3 Limitations of multi-threading

There is an ongoing debate on the necessity of running a large number of threads in a single process. Proponents argue with simpler implementations of e.g. one thread per connection for network severs whereas critics dislike the abstraction of problems by creating a huge number of threads. We do not discuss this issue here but focus on the factors that limit the number of threads in commonly used runtime systems.

Newly created threads share most of the resources of the process but differ in the execution context, i.e. the set of registers to represent the state. One important resource controlled by the CPU is the runtime stack. The number of threads that can be created

is limited by this resource. This is because of the part of address space that has to be allocated for each thread's runtime stack. Since threads in a multi-threaded process share one address space and the reserved address space for a thread cannot grow they have to be reserved large enough in order that they do not overflow as they grow because todays compilers require contiguous memory to be used for the stack. However, recent work on a split stack for GCC is in progress.[8]

Hence, the reserved address space for a thread's runtime stack determines the maximum size of the stack. This is an internal fragmentation problem on the address space of a process. Choosing a too large initial size of the stack increases the fragmentation significantly whereas choosing a too small stack size limits e.g. the recursion depth of a thread and a stack overflow is more likely. Typically the initial size can be chosen by the programmer but it is very hard and often impossible to make a worst-case stack size estimation. Also a lot of programmers do not even care about this problem. High level languages such as Java or C# provide easy to use methods to create multi-threaded programs but suffer from the same limitations.

The default initial size of a thread's stack depends on the implementation and architecture. For most 32-bit architectures the stack size for a POSIX thread is 2 MiB. Given the userspace address space of 3 GiB that the x86 Linux Kernel provides, this yields about 1200 threads under good circumstances.[9] As a result there is no address space left for dynamic memory allocation. If e.g. a network server process starts one thread per connection such an implementation will quickly run out of free address space. Typically a network connection can be represented with less then 2 MiB and therefore the largest part of the reserved address space is unused. Note that a small stack which needs only a few kilobytes also only uses a few pages of memory on system that supports demand paging which most operating systems do. The problem is the waste of address space and not the waste of physical memory. New 64-bit hardware facilitates the problem but does not solve it either.

3.4 Segmented stacks

Go is a language designed to support thousands or even millions of different threads of execution (goroutines) in one address space. To achieve such multi-threading support in the runtime system the limitations described above has to be neutralized. This is done by removing the constraint that a runtime stack must exist in contiguous memory. Such a stack implementation is called *segmented stack*. Instead of having one contiguous block of memory the stack is represented by a linked list of stack segments. Hence, the stack of a goroutine start with a small amount of memory and can grow on demand. The Go compiler generates Code to check the bounds of the current runtime stack and adds or removes the links as needed. Note that at the point of writing, this is only supported by the *6g* or *8g* Go-compilers, but support for GCC is on the way.[8] [10]

4 Concurrency

The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes. In some concurrent computing systems, communication between the concurrent components is hidden from the programmer (e.g., by using futures), while in others it must be handled explicitly. Explicit communication can be divided into two classes:

Shared memory communication:[11]
Concurrent components communicate by altering the contents of shared memory locations (exemplified by Java and C#). This style of concurrent programming usually requires the application of some form of locking (e.g., mutexes, semaphores, or monitors) to coordinate the communication between threads.

Message-passing communication:[11]
Concurrent components communicate by exchanging messages (exemplified by Go and Erlang). The exchange of messages may be carried out asynchronously, or may use a rendezvous style in which the sender blocks until the message is received. Asynchronous message passing may be reliable or unreliable (sometimes referred to as "send and pray"). Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming. A wide variety of mathematical theories for understanding and analyzing message-passing systems are available, including the *Actor model*[1], and various process calculi. Message passing can be efficiently implemented on symmetric multiprocessors, with or without shared coherent memory.

Thinking of Java or other similar object oriented languages (using shared memory communication) concurrency and multi-threaded programming increases difficulty enormously. Typically shared memory communication is achieved by global variables. This effects the programmer to take care for a lot of side-effects (for example race conditions).

[1] In computer science, the Actor model is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation. An actor can make local decisions in response to a message that it received.[12].

4.1 Share by communicating

In Go concurrency plays an important role. Therefor it has explicit support for concurrent programming. For thread communication it takes advantage of message passing.

The main-slogan for Go's concurrency-concept is:

> *Do not communicate by sharing memory; instead, share memory by communicating*[2]

This communication is done by channels. Passing shared memory around on channels effects that thread-communication becomes simple and safe. All known side-effects of sharing memory by global variables are avoided. For example there is no need to put a mutex around a variable because only one goroutine has access to the channel at any given time.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It does not require any synchronization primitives. Now run another such instance. There is no need for synchronization either. Now let those two communicate. If the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, perfectly fit this model. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

4.2 Goroutines

Goroutines are functions executing in parallel with other goroutines in the same address space. A running program consists of one or more goroutines. It's not the same as a thread, coroutine, process, etc. but the name alludes to coroutines. In general, goroutines are much more powerful than coroutines. In particular, it is easy to port coroutine logic to goroutines and gain parallelism in the process.

Goroutines are created with the `go` statement from anonymous or normal functions. This is very easy in contrast to Java where you have to create a class which implements the Runable interface or extends the Thread class. Goroutines exit by returning from their toplevel function, or just falling off the end. There is also the possibility to use `runtime.Goexit()` inside the code.

Go uses small lightweight threads for running goroutines in parallel. They do not necessarily run in separate operating system threads, but a group of goroutines are multiplexed onto multiple threads. Execution control is moved between them by blocking them when sending or receiving messages over channels. When a goroutine executes a blocking system call, no other goroutine is blocked.

Using goroutines and channels makes concurrency easy to use. It hides many of the complexities of thread creation and management. In the following code-example we execute two expensive compuataions in parallel. We demonstrate the Java-code for this simple task and afterwards the similar Go-code.

Listing 4.1: Parallel computation in Java

```
1   public static void main(String[] args) {
2       new ParallelComputationWrapper().start();
3       anotherExpensiveComputation(a, b, c);
4   }
5
6   public class ParallelComputationWrapper extends Thread {
7       public void run() {
8           expensiveComputation(x, y, z);
9       }
10  }
```

Listing 4.2: Parallel computation in Go

```
1   func main() {
2       go expensiveComputation(x, y, z)
3       anotherExpensiveComputation(a, b, c)
4   }
```

Listing 4.3 shows a parallel computation example taking use of an anonymous function. The calculation steps `expensiveComputationStep1` to `expensiveComputationStep3` are performed sequential but in parallel with `anotherExpensiveComputation`.

Listing 4.3: Parallel computation using an anonymous function

```
1   func main() {
2       go func() {
3           expensiveComputationStep1(x, y, z)
4           expensiveComputationStep2(x, y, z)
5           expensiveComputationStep3(x, y, z)
6       }()
7       anotherExpensiveComputation(a, b, c)
8   }
```

Within a single goroutine, reads and writes must behave as if they were executed in the order specified by the program. That is, compilers and processors may reorder the reads and writes executed within a single goroutine only when the reordering does not change the behavior within that goroutine as defined by the language specification. Because of this reordering, the execution order observed by one goroutine may differ from the order perceived by another. For example, if one goroutine executes `a = 1; b = 2`, another might observe the updated value of `b` before the updated value of `a`.

4.2.1 Once

Let's take a look on the following bad example (Listing 4.4) where calling `twoprint` should cause "hello, world" to be printed twice.

Listing 4.4: A bad synchronization example

```
1  var a string
2  var done bool
3
4  func setup() {
5      a = "hello, world"
6      done = true
7  }
8  func doprint() {
9      if !done {
10         setup()
11     }
12     print(a)
13 }
14 func twoprint() {
15     go doprint()
16     go doprint()
17 }
```

There is no guarantee that, in `doprint`, observing the write to `done` implies observing the write to `a`. This version can incorrectly print an empty string instead of "hello, world". Furthermore there is no guarantee that setup runs only once. To fix this second issue we need a mutex inside the `doprint` function.

The package *once* provides a safe mechanism for initialization in the presence of multiple goroutines. For one-time initialization that is not done during init, we have to wrap the initialization in a niladic[2] function `f` and call `once.Do(f)`. Multiple threads can execute `once.Do(f)` for a particular `f`, but only one will run `f()`, and the other calls block until `f()` has returned.

Let's fix our `twoprint`-example using the once package. Now the first call to `doprint` runs `setup` once and guarantees the initialization. By the way, we'll get a much clearer code.

[2] A function (or operator) is niladic if it takes no arguments

Listing 4.5: Using once for initialization

```
1  var a string
2
3  func setup() {
4      a = "hello, world"
5  }
6  func doprint() {
7      once.Do(setup)
8      print(a)
9  }
10 func twoprint() {
11     go doprint()
12     go doprint()
13 }
```

4.3 Channels

As mentioned above shared values are passed around on channels. Each send on a particular channel is matched to a corresponding receive from that channel, usually in a different goroutine. The operater `<-` is used for reading and writing on channels. Depending on whether the channel is left or right of the operator, the channel is written or read. Like maps, channels are a reference type and are allocated with make. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

Listing 4.6: Allocating channels

```
1 ci := make(chan int)            // unbuffered channel of integers
2 cj := make(chan int, 0)         // unbuffered channel of integers
3 cs := make(chan *os.File, 100)  // buffered channel of file-pointers
```

Channels show the following blocking behavior:

1. The receivers always block until there is data to receive.
2. If the channel is unbuffered, the sender blocks until the receiver has received the value.
3. If the channel has a buffer, the sender blocks if the buffer is full.

Due to this natural blocking behavior you can make one goroutine wait for an other. This fact let *channels combine communication with synchronization.*

In the following example (Listing 4.7) we show how to use a channel for synchronization. For that we want to sort a list in parallel while doing some other computation. At a certain point we want to use the sorted list.

Listing 4.7: Using a channel for synchronization

```
1  c := make(chan int)  // Allocate a channel.
2
3  // Start the sort in a goroutine
4  // When it completes, signal on the channel.
5  go func() {
6      list.Sort()
7      c <- 1  // Send a signal; value does not matter.
8  }()
9
10 doSomethingForAWhile()
11 <- c        // Wait for sort to finish; discard sent value.
12 doSomethingWithTheSortedList()
```

Next, we show how channels can be used to combine communication with synchronization. Therefore we extend our parallel computation example from Listing 4.2 using a channel.

Listing 4.8: Parallel computation using a channel

```
1  func computeAndSend(x, y, z int) chan int {
2      ch := make(chan int)
3      go func() {
4          ch <- expensiveComputation(x, y, z)
5      }()
6      return ch
7  }
8  func main() {
9      ch := computeAndSend(x, y, z)
10     v2 := anotherExpensiveComputation(a, b, c)
11     v1 := <-ch
12     fmt.Println(v1, v2)
13 }
```

A buffered channel can be used like a semaphore, for instance to limit throughput. We will show this later with an example.

4.3.1 Channels of channels

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. This implies that cannels can be passed around on other channels. For showing the importance of this language-detail let's think of a request-broker. The requests should be processed in parallel. We obtain a request object from the client. Inside this request object there is a channel on which

to receive the response. Then we need a second channel for sharing the request object to the worker-goroutine. So we have to pass the response-channel (which is inside the request object) through the second channel to the worker-goroutine.

4.4 Parallelization

If the calculation can be broken into separate pieces, it can be parallelized to speed up the computation on multiple CPUs. As already shown we can use a channel to signal when each piece completes.

Current compiler implementations (6g, 8g, ...) do not parallelize code by default.[2] There are two ways to perform calculations in parallel on multiple CPUs. One can use the environment variable `GOMAXPROCS` to set the number of cores to use or call `runtime.GOMAXPROCS(NCPU)` explicitly from inside the code.

4.4.1 Futures

Sometimes one knows that the computation of a value is needed before this value is actually needed. In this case, one can potentially start computing the value on another processor and have it ready when it is needed. This is the idea behind futures. Futures are easy to implement via closures and goroutines.[13]

As an example we write a function which takes two matrices, computes the inverse from both and return the product of the two inverse matrices. The inverse computations can be done in parallel.

Listing 4.9: Computing inverse matrices in parallel

```
1  func InverseProduct (a Matrix, b Matrix) {
2      a_inv_future := InverseFuture(a);
3      b_inv_future := InverseFuture(b);
4      a_inv := <-a_inv_future;
5      b_inv := <-b_inv_future;
6      return Product(a_inv, b_inv);
7  }
8  func InverseFuture (a Matrix) {
9      future := make (chan Matrix);
10     go func () { future <- Inverse(a)  }();
11     return future;
12 }
```

The `InverseFuture` function launches a anonymous goroutine to perform the inverse computation and immediately returns a channel on which the caller can read the result.

Naturally the read on the channel blocks until the computation is done but both computations are started before the first read blocks. The Inverse is computed asynchronously and potentially in parallel.

4.4.2 Generators

A generator is a special functions that return the next value in a sequence each time the function is called. Generators can be used to control the iteration behavior of a loop. Mostly they are used to introduce parallelism into loops. Generators in Go are implemented with goroutines, though in other languages coroutines are often used.[13] If the generator's task is computationally expensive, the generator pattern can allow a consumer to run in parallel with a generator as it produces the next value to consume.

For example, we create an iterator over random numbers. The random number generator can produce the numbers in parallel. Of course the numbers are passed via a channel. This effects the number-consumer to wait until the next number is generated (if it isn't already) but for the producer there is no need to wait with generating the next number.

Listing 4.10: Random number generator

```
1  func generateRandomNumbers (n int) {
2      ch := make (chan float)
3      sem := make (semaphore, n)
4
5      for i := 0; i < n; i++ {
6          go func () {
7              ch <- rand.Float()
8              close(ch)
9          } ()
10     }
11
12     // launch extra goroutine to close channel
13     go func () {
14         sem.Wait(n)
15         close(ch)
16     }
17
18     return ch
19 }
```

Listing 4.11: Consumer of ramdom numbers

```
1  for x := range generateRandomNumbers(100) {
2      fmt.Println(x) // do whatever you want ;)
3  }
```

4.4.3 Parallel For-Loop

Parallel for-loops can be used if each iteration (the loop body) can be calculated independent of other runs. In this case we can increase performance by parallelization but the iterator-variable has to be shared. Go doesn't support parallel for-loops as a separate construct, but they are easy to implement using goroutines.[13]

For example we show a loop for scalar addition on a vector.

Listing 4.12: Consumer of ramdom numbers

```
1  func VectorScalarAdd (v []float, s float) {
2      sem := make (semaphore, len(v));
3      for i,_ := range v {
4          go func (i int) {
5              v[i] += s;
6              sem.Signal();
7          } (i);
8      }
9      sem.Wait(len(v));
10 }
```

When implementing a function which contains one big parallel for-loop (like the VectorScalarAdd example above), one can increase parallelism by returning a future rather than waiting for the loop to complete. This way you can start the VectorScalarAdd-function in parallel with other computation.

4.4.4 Semaphores

A semaphore usually takes one of two forms:[14] binary and counting. A binary semaphore is a simple flag that controls access to a single resource. A counting semaphore is a counter for a set of available resources. Semaphores can be used to implement mutexes, limit access to multiple resources, solve the readers-writers problem, etc.

There is no semaphore implementation in Go's sync package, but they can be emulated easily using buffered channels.[13] The capacity of the buffered channel represents the set of available resources. The length of the channel is the number of resources currently being used. We do not care about what is stored in the channel, only its length.

In our closing example we show a counting semaphore on buffered channels.

4.4.5 Example

A good example for parallelization is a request-broker [15]. We want to handle a defined number of requests in parallel and block incoming requests if the maximum number is reached. Therefor we use a buffered channel like a semaphore. The statement `sem<-1` acquires the semaphore and `<-sem` releases it. Within this section, the request is processed.

The server should never be blocked and must be able to accept all requests nearly immediately. We achieve this by spawning a new goroutine (`go handle(req)`) as soon as we got the request object from the request-queue channel. The request-queue channel blocks if there is no outstanding request.

Listing 4.13: A request-broker in Go

```
1  var sem = make(chan int, MaxOutstanding)
2  func handle(r *Request) {
3      sem <- 1;     // Wait for active queue to drain.
4      process(r);   // May take a long time.
5      <-sem;        // Done; enable next request to run.
6  }
7  func Serve(queue chan *Request) {
8      for {
9          req := <-queue;
10         go handle(req);  // Don't wait for handle to finish.
11     }
12 }
```

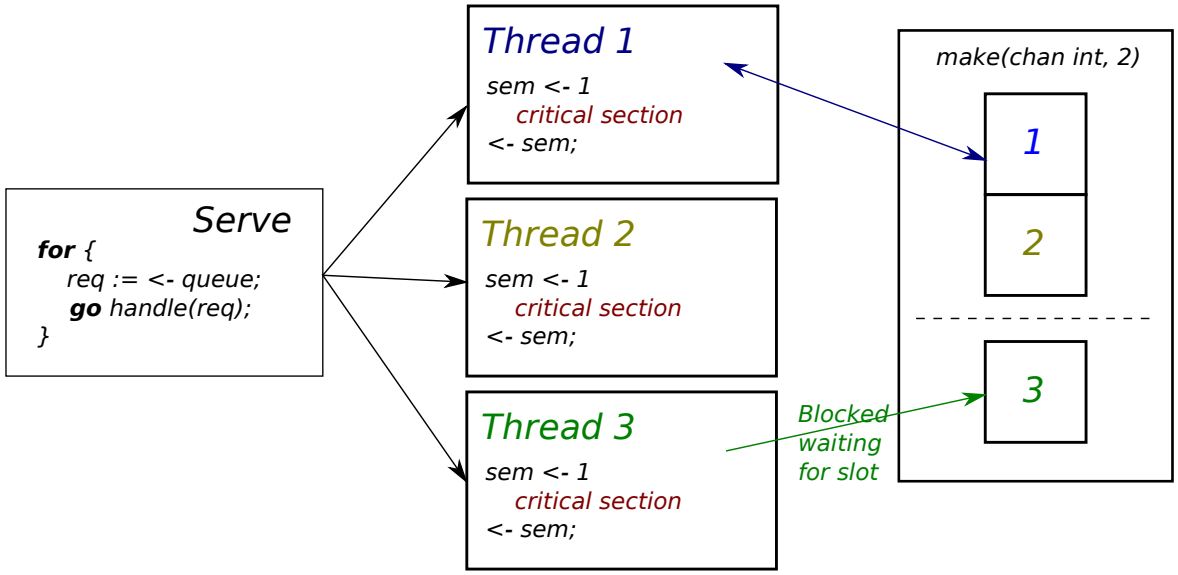


Figure 4.1: Channel of length 2 blocking the third request (thread)

Bibliography

[1] R. Cox. research!rsc blog, 2009. URL `http://research.swtch.com/2009/11/go-data-structures.html`.

[2] The Go programming language, 2010. URL `http://golang.org`.

[3] Duck typing. Wikipedia, the free encyclopedia, 2010. URL `http://en.wikipedia.org/wiki/Duck_typing`.

[4] R. Pike. Another Go at language design. Stanford University Computer Systems Laboratory Colloquium, April 2010. URL `http://www.stanford.edu/class/ee380/`.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proc. OOPSLA*. ACM, 2004.

[6] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL*. ACM, 2003.

[7] G. Kliot, E. Petrank, and B. Steensgaard. A lock-free, concurrent, and incremental stack scanning mechanism for garbage collectors. *SIGOPS Oper. Syst. Rev.*, 43(3):3–13, 2009.

[8] I. L. Taylor. Split stacks in GCC, September 2009. URL `http://gcc.gnu.org/wiki/SplitStacks`.

[9] U. Drepper. Thread numbers and stacks, October 2005. URL `http://people.redhat.com/drepper/thread-number-stacks.html`.

[10] GCC Steering Commitee announcement to support Go, 2010. URL `http://article.gmane.org/gmane.comp.gcc.devel/111603`.

[11] Concurrent computing. Wikipedia, the free encyclopedia, 2010. URL `http://en.wikipedia.org/wiki/Concurrent_computing`.

[12] Actor model. Wikipedia, the free encyclopedia, 2010. URL `http://en.wikipedia.org/wiki/Actor_model`.

[13] Go language patterns, 2010. URL `http://sites.google.com/site/gopatterns/`.

[14] Semaphore. Wikipedia, the free encyclopedia, 2010. URL `http://en.wikipedia.org/wiki/Semaphore_(programming)`.

[15] Go L4, 2010. URL `http://www.technovelty.org/code/go-l4.html`.