dev.cc branch plan
(golang.org/s/dev.cc)

Russ Cox
November 2014

This document serves as supporting documentation for the CLs made in the dev.cc branch, to avoid repeating that information in each CL. It also holds the list of TODOs identified during the code review process. The plan is to get everything running first and clean things up second.

The goal of the dev.cc branch is to eliminate all C code from the Go libraries and then remove the C compiler that we have been maintaining as part of the toolchain (the cmd/5c, cmd/6c, cmd/8c, and cmd/cc directories). As of Go 1.4, the only C code compiled with that compiler is in package runtime. Therefore the work involved is mainly the conversion of package runtime completely to Go.

There is a separate effort to rewrite the Go compiler (the cmd/5g, cmd/6g, cmd/8g, cmd/gc directories) in Go, but that is unrelated. That code compiles with the host compiler (gcc or clang); it does not depend on cmd/5c etc.

A full list of CLs is at the end of the document.

runtime Go conventions (or lack thereof)

The most obvious requirement is that all the src/runtime/*.c files need to become *.go files. I have been converting these using a variant of the C->Go translator I built for converting the Go compiler. The contexts are sufficiently different that it doesn't do a perfect job, but it's a better starting point than the original files. I edit the result by hand to fix things it doesn't convert well (such as pointer arithmetic).

In the initial conversion, all type names are lowercased completely: the C struct GCStats becomes the Go type gcstats, and so on. This creates conflicts with variable names. These have been adjusted in an ad-hoc fashion so far, and I'm not yet going to try to clean up what's already happened. A list of conflicts follows. Keep in mind that the intent is to clean up them all later.

- the G struct and the g global thread-local variable. In many by-hand conversions g was renamed to gp, but this violates the previous convention in the runtime package that g was always the local goroutine and gp was always some other goroutine. In Go, the g thread-local variable must be read using the function call getg(), but then a name must be given to the result. Using "g := getg()" breaks code that goes on to refer to the type (G, now g). Using "gp := getg()" breaks code that already has a gp variable. For the

conversions that remain, We use "_g_ := getg()". This is obviously ugly, but it avoids both of those problems and it will be very easy to find and clean up later.

- the M struct and variables m of type *M. Same as with G/g: the m variable shadows the m type, breaking code that tries to refer to the m type later. We use _m_ when necessary.
- the P struct and variables p of type *P. Same. We use _p_ when necessary.
- the MStats struct and the mstats global variable. C had a #define mstats memstats, so now the Go code always refers to 'memstats' for the variable.
- the MHeap struct and the mheap global variable. C had a #define mheap meap_ [sic], so now the Go code always refers to 'mheap_' for the variable.

The current runtime contains auto-generated Go definitions for C constants. The rule in the converter is that an upper case C constant like PageSize is prefixed by an underscore, producing the Go const _PageSize. Some ad-hoc conversions had already introduced separate definitions for initial-lower-cased names like pageSize. When the auto-generator was added, we rewrote those definitions to say things like "pageSize = _PageSize" so that there was still just one point of truth (the C sources). The newly converted files use mostly the _ form, but in some of the malloc code where the lower case form is more prevalent, We changed the script to use those. Regardless, no attempt is being made right now to settle on a single form. Like the type-vs-variable name conflicts, they will be cleaned up later.

The initial conversion is meant to proceed a file at a time, so that the C and Go equivalent code can be diffed easily. Because some files have already been partially converted, this means we end up with multiple files that should be a single file. For now we just keep creating new files by numbering them: runtime.c becomes runtime1.go, runtime2.go; then runtime.h becomes runtime3.go. And so on. The code will be rearranged into better file boundaries (and ideally package boundaries) later.

runtime signal handling

The Unix signal handling code makes unfortunate use of various #define macros to approximate portability. This code needs to be rewritten to use simple functions instead. Since the Go compiler can be counted on to inline them, the performance will be comparable (not that it really matters) and the code will be cleaner. But the code is not a direct line-for-line translation.

assembler help

The current runtime contains assembly header files (zasm_darwin_amd64.h and so on) auto-generated from the C sources. These files give #defines for struct field offsets and constant values. When we remove the C sources, we will still have assembly files that need this information, so it must be generated by the Go compiler instead.

We will change cmd/gc to add a -asmhdr flag that causes it to write an assembly header to the named file. Assembly files can #include that instead of zasm_darwin_amd64.h. We will change cmd/dist and cmd/go to arrange for writing of an assembly header to “go_asm.h” in the per-package work directory before invoking the assembler on any assembly files. The assembly files will say ‘#include “go_asm.h”’. There is a little bit of TLS-specific information that cmd/dist adds to the assembly header as well. That will move out of cmd/dist into a separate file “src/runtime/go_tls.h”.

cmd/go could use the -asmhdr flag to provide a “go_asm.h” for any package containing assembly, not just package runtime. It could be limited to package runtime but it’s not clear why we should.

cgo help

C is also used outside the runtime in the implementation of cgo wrappers. Cgo writes C files in order to get at two features provided by the C compiler: #pragmas that turn into directives passed along to the linker, and the ability to control the fully qualified “linker” symbol name (for example, plain X instead of import/path.X). These features need to be provided by the Go compiler instead.

Each current #pragma cgo_xxx accepted by the C compiler will turn into a //go:cgo_xxx comment accepted by the Go compiler.

To control the linker symbol name, the Go compiler will also accept a comment of the form

```
//go:linkname <local-name> <link-name>
```

to specify that the global variable named by <local-name> should be referred to as <link-name> in the object files being written. For example, in order to define a uint32 usable by gcc-compiled C code as go_var_x, one can write:

```
package p
//go:linkname anything go_var_x
var anything uint32
```

Assuming p is imported as just “p”, the default linkname for var anything would be p.anything. The //go:linkname directive changes it to go_var_x. Most often the important part is dropping the import path prefix, since ordinary gcc-compiled C code cannot refer to names containing . or /. Note that this makes go:linkname meaningful even if the variable were also named go_var_x. That is, in this code snippet:

```
package p
//go:linkname go_var_x go_var_x
```

```
var go_var_x uint32
```

the //go:linkname line is NOT a no-op. It is forcing the linkname of the Go variable to omit the usual import path prefix.

This control over linker name also allows a package to name symbols in other packages without resorting to assembly files. The current src/runtime/thunk.s can be replaced with uses of this mechanism instead, which will be easier to maintain. However, because of the possible abstraction violation that can be done with //go:linkname, we may choose to limit its use to certain packages or source files. To start, we require that a source file that contains a //go:linkname line also imports “unsafe”.

cmd/dist changes

For the most part, cmd/dist just gets simpler. The Go generated from C goes away. The -asmhdr flag on the Go compiler will write out exactly what is needed, instead of cmd/dist having to massage a different output format from the C compiler. The only real generated code it needs to write is things like the current version and experiment strings and maybe some constants about which GOOS and GOARCH are in use.

I think we can get to the point where cmd/dist writes no GOOS+GOARCH-specific files (only GOOS-specific or GOARCH-specific). That would let us have “go tool dist bootstrap” just write all the possible files during the first make.bash, with no new make.bash invocations required to start cross-compiling.

TODO

This section lists work left to be done in future CLs, other than the work already mentioned above. It is a place to record suggestions made during the code review process that cannot be done right then.

- restore GOEXPERIMENT support
- stop generating zsys_GOOS_GOARCH.s; it’s only used on Windows and can be done in runtime directly with go generate. then there are no GOOS-GOARCH-specific files.
- generate all GOOS-specific files during dist, not just the current GOOS
- generate all GOARCH-specific files during dist, not just the current GOARCH
- in signal code, drop _NSIG in favor of len(sigtable)

CLs

(Each is at https://codereview.appspot.com/#########.)

170320044 build: disable API check until all systems build

169330045 cmd/cgo: generate only Go source files
172960043 cmd/dist: adjust for build process without cmd/cc
169360043 cmd/gc: changes for removing runtime C code
171470043 cmd/go: adjust go, cgo builds & disable cc
171480043 liblink: resolve bss vs other conflict regardless of order found
167520044 reflect: interfaces contain only pointers
168500043 runtime/cgo: convert from C to Go
174860043 runtime: convert arch-specific .c and .h files to Go
168510043 runtime: convert assembly files for C to Go transition
170330043 runtime: convert basic library routines from C to Go
171490043 runtime: convert defs_$GOOS_$GOARCH.h to Go
167550043 runtime: convert header files to Go
167540043 runtime: convert memory allocator and garbage collector to Go
174830044 runtime: convert operating system support code from C to Go
166520043 runtime: convert panic and stack code from C to Go
172250043 runtime: convert parallel support code from C to Go
172250044 runtime: convert race implementation from C to Go
172260043 runtime: convert scheduler from C to Go
168500044 runtime: convert signal handlers from C to Go