Dominik Honnef

Are you looking for a Go programmer to implement your latest idea?
Send me an email!

Statically compiled Go programs, always, even with cgo, using musl

(Click here to skip the flavor text and go straight to the commands.)

Most people appreciate the fact that Go programs compile statically, as it makes deploying Go programs a lot easier. Unfortunately, they aren't always statically compiled.

Using cgo, either in a 3rd party package or in the standard library, usually requires linking to a libc. Both `net` and `os/user` are two packages known for using cgo and requiring a libc. On most Linux systems, that leaves us with programs dynamically linking against the glibc. Usually, that's not a problem in itself, if we deploy our programs to machines that have a glibc of a matching version. If they don't have a matching glibc, however, things tend to go south.

The first solution that may come to mind is to just link the glibc statically. However, that rarely works, as various warnings will tell you. The glibc just doesn't like that. Also, it might create certain legal problems. Some people, the FSF in particular, interpret static linking as copying, and thus any binaries linked statically with the glibc that we distribute would have to abide by the GPL.

The second solution, which most people go for, is to avoid cgo altogether. They use the `netgo` tag to build a pure-Go version of the `net` package, and they set CGO_ENABLED=0. Of course this has the drawback of not being able to use cgo. That means you can't even use `os/user` to get the name of the current user.

Luckily, there is a third solution: Don't use the glibc. The glibc isn't the only libc implementation on Linux. There are others, and musl is one of them. It's a lightweight libc (it's a lot smaller than the glibc) which supports all the relevant features, was written with static linking and correctness in mind, and is licensed under the MIT. It seems to be the perfect fit for our requirements.

Installing musl

Installing musl is quite straightforward and follows the usual 3 steps: configure, make, make install:

```
$ wget http://www.musl-libc.org/releases/musl-1.1.10.tar.gz
$ tar -xvf musl-1.1.10.tar.gz
$ cd musl-1.1.10
$ ./configure
$ make
$ sudo make install
```

This will install musl in `/usr/local/musl` – it will not conflict with your glibc. Alternatively, you can pass `--prefix` to `./configure` to install it elsewhere; for example in your home directory.

musl comes with a wrapper script called `musl-gcc`, which invokes gcc with all the options required for using musl. We can test it on a pure C program:

```
$ cat > hello.c <<EOF
#include <stdio.h>
int main()
{
        printf("hello, world!\n");
        return 0;
}
EOF
$ /usr/local/musl/bin/musl-gcc -static hello.c
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
$ ./a.out
hello, world!
```

We end up with a 12 KB large, statically compiled program. No glibc in sight.

Using it with Go is a bit more complex, but still simple enough:

```
$ cat > hello.go <<EOF
package main

// #include <stdio.h>
// void helloworld() { printf("hello, world\n"); }
import "C"

import (
        "log"
        "net"
)

func main() {
        log.Println(net.LookupHost("google.com"))
        C.helloworld()
}
EOF

$ CC=/usr/local/musl/bin/musl-gcc go build --ldflags '-linkmode external -extldflags "-static"' hello.go
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
$ ./hello
2015/06/22 06:01:40 [216.58.211.14 2a00:1450:4016:804::200e] <nil>
hello, world
```

We have a fully static binary that is using musl's DNS resolver[1], as well as a custom C function. The output binary is 2.8 MB large. How large would it be if we linked dynamically instead? Still 2.8 MB. The musl libc is quite small and its maximum overhead will be about 450 KB – that is, if you use all of it, which you are unlikely to.

But back to the actual command and what we're doing here:

1. We set the CC environment variable. This tells Go which compiler and linker to use, in our case the musl-gcc wrapper
2. We use `--ldflags` to pass two options to Go's linker:
    1. `-linkmode external` instructs Go's linker to always use an external linker, that is gcc (more about this later)
    2. `-extldflags "-static"` instructs Go's linker to pass the `-static` flag to the external linker

These 3 settings are all that is required to use musl with Go.

Do note that our basic musl setup only works for cgo that uses the C standard library. It will not work for external dependencies. For example, if we wanted to use `github.com/hanwen/usb` – bindings for libusb – we'd also have to build libusb with musl first. An alternative approach here would be to have a VM that uses one of the musl-based Linux distributions, such as Alpine Linux. That way, we don't have to deal with compiling those libraries.

Drawbacks

Unfortunately, there are some minor drawbacks to using musl. The first one is compatibility. Earlier we said that musl was written with correctness in mind. The problem is that some software out there isn't correct, and won't build with musl. On the one hand, the glibc is a lot more forgiving and will accept certain code that it shouldn't. On the other hand, the glibc comes with its own custom extensions. If code depends on these, it will not work with musl.

Luckily, this sounds worse than it is. Thanks to several Linux distributions using musl as their default libc, a lot of these issues are being fixed. Also, they most often affect large applications, not libraries like we might use with cgo.

Another drawback is that while musl supports C++, it doesn't so as easily as with C. We can't just use the musl-gcc wrapper, but instead would need to build a libstdc++ with musl. It's doable, but outside the scope of this article. Unfortunately, the lack of easy C++ support also means that we can't use Go's race detector with musl, as it is written in C++.

Yet another drawback is that the `os/user` package currently doesn't work with musl. It's a bug that is easily fixable and I hope that it will be fixed in Go 1.6. In the meantime, you can apply a patch I've included at the end of this article.

The final drawback is compilation speed. Remember that `-linkmode` flag? When you're not using cgo in any 3rd party packages, but only due to the standard library (`net`, `os/user` and so on), Go doesn't normally use an external linker. Instead, it does all the linking itself. Forcing it to use the external linker means that we now have to invoke gcc, which will be a bit slower. It's a minor slowdown, though. For a simple hello world program, the difference between internal and external linking is roughly 0.09ms (0.24ms vs 0.33ms). It won't kill us.

Enjoy your static binaries!

[1]: Do note that musl doesn't support NSS, so using musl's DNS resolver is not much different from using netgo: You won't be able to use things like mDNS.

The os/user patch

```
--- c/src/os/user/lookup_unix.go
+++ w/src/os/user/lookup_unix.go
@@ -9,7 +9,6 @@ package user

 import (
        "fmt"
-       "runtime"
        "strconv"
        "strings"
        "syscall"
@@ -55,17 +54,15 @@ func lookupUnix(uid int, username string, lookupByName bool) (*User, error) {
        var pwd C.struct_passwd
        var result *C.struct_passwd

-       var bufSize C.long
-       if runtime.GOOS == "dragonfly" || runtime.GOOS == "freebsd" {
-               // DragonFly and FreeBSD do not have _SC_GETPW_R_SIZE_MAX
-               // and just return -1.  So just use the same
-               // size that Linux returns.
+       bufSize := C.sysconf(C._SC_GETPW_R_SIZE_MAX)
+       if bufSize == -1 {
+               // DragonFly and FreeBSD do not have _SC_GETPW_R_SIZE_MAX.
+               // Additionally, not all Linux systems have it, either. For
+               // example, the musl libc returns -1.
                bufSize = 1024
-       } else {
-               bufSize = C.sysconf(C._SC_GETPW_R_SIZE_MAX)
-               if bufSize <= 0 || bufSize > 1<<20 {
-                       return nil, fmt.Errorf("user: unreasonable _SC_GETPW_R_SIZE_MAX of %d", bufSize)
-               }
+       }
+       if bufSize <= 0 || bufSize > 1<<20 {
+               return nil, fmt.Errorf("user: unreasonable _SC_GETPW_R_SIZE_MAX of %d", bufSize)
        }
        buf := C.malloc(C.size_t(bufSize))
        defer C.free(buf)
```

Related articles

- 01 Dec 2014 An incomplete list of Go tools
- 16 Apr 2016 Merging multiple git repositories

Monday, June 22, 2015 - by Dominik Honnef

Github
Twitter

Email me