Calling a Windows DLL

Go allows you to call native Windows function in several different ways.

1. Dynamically load a DLL, then call a function in it. You can call the function via `SyscallX` (where X is the number of parameters. If the function has fewer parameters than that, for example passing 7 arguments to a function that accepts 9, `Syscall9` will still work, you just need to specify 7 as your second argument to `Syscall9`).

A sample Go program that calls a Windows DLL function using this method:

```
package main

import (
    "fmt"
    "syscall"
    "unsafe"
)

func abort(funcname string, err error) {
    panic(fmt.Sprintf("%s failed: %v", funcname, err))
}

var (
    kernel32, _         = syscall.LoadLibrary("kernel32.dll")
    getModuleHandle, _ = syscall.GetProcAddress(kernel32, "GetModuleHandleW")

    user32, _      = syscall.LoadLibrary("user32.dll")
    messageBox, _ = syscall.GetProcAddress(user32, "MessageBoxW")
)

const (
    MB_OK                = 0x00000000
    MB_OKCANCEL          = 0x00000001
    MB_ABORTRETRYIGNORE  = 0x00000002
    MB_YESNOCANCEL       = 0x00000003
    MB_YESNO             = 0x00000004
    MB_RETRYCANCEL       = 0x00000005
    MB_CANCELTRYCONTINUE = 0x00000006
    MB_ICONHAND          = 0x00000010
    MB_ICONQUESTION      = 0x00000020
    MB_ICONEXCLAMATION   = 0x00000030
    MB_ICONASTERISK      = 0x00000040
    MB_USERICON          = 0x00000080
    MB_ICONWARNING       = MB_ICONEXCLAMATION
    MB_ICONERROR         = MB_ICONHAND
    MB_ICONINFORMATION   = MB_ICONASTERISK
    MB_ICONSTOP          = MB_ICONHAND

    MB_DEFBUTTON1 = 0x00000000
    MB_DEFBUTTON2 = 0x00000100
```

```
    MB_DEFBUTTON3 = 0x00000200
    MB_DEFBUTTON4 = 0x00000300
)

func MessageBox(caption, text string, style uintptr) (result int) {
    var nargs uintptr = 4
    ret, _, callErr := syscall.Syscall9(uintptr(messageBox),
        nargs,
        0,
        uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(text))),
        uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(caption))),
        style,
        0,
        0,
        0,
        0,
        0)
    if callErr != 0 {
        abort("Call MessageBox", callErr)
    }
    result = int(ret)
    return
}

func GetModuleHandle() (handle uintptr) {
    var nargs uintptr = 0
    if ret, _, callErr := syscall.Syscall(uintptr(getModuleHandle), nargs, 0, 0, 0);
callErr != 0 {
        abort("Call GetModuleHandle", callErr)
    } else {
        handle = ret
    }
    return
}

func main() {
    defer syscall.FreeLibrary(kernel32)
    defer syscall.FreeLibrary(user32)

    fmt.Printf("Return: %d\n", MessageBox("Done Title", "This test is Done.",
MB_YESNOCANCEL))
}

func init() {
    fmt.Print("Starting Up\n")
}
```

2. Using syscall.NewProc instead of syscall.GetProcAddress. These are basically some helper methods over the syscall ones, you saw above, and are available in Windows only: http://golang.org/src/pkg/syscall/dll_windows.go

```
package main

import (
    "fmt"
    "syscall"
    "unsafe"
)

func main() {
    var mod = syscall.NewLazyDLL("user32.dll")
    var proc = mod.NewProc("MessageBoxW")
    var MB_YESNOCANCEL = 0x00000003

    ret, _, _ := proc.Call(0,
        uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr("This test is Done."))),
        uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr("Done Title"))),
        uintptr(MB_YESNOCANCEL))
    fmt.Printf("Return: %d\n", ret)

}
```

3. By "linking" against the library, using the "[[cgo]]" method (this way works in Linux and Windows). Example:

```
import ("C")
...
C.MessageBoxW(...)
```

See [[cgo]] for further details.