# Building `sys/unix`

The sys/unix package provides access to the raw system call interface of the underlying operating system. See: https://godoc.org/golang.org/x/sys/unix

Porting Go to a new architecture/OS combination or adding syscalls, types, or constants to an existing architecture/OS pair requires some manual effort; however, there are tools that automate much of the process.

## Build Systems

There are currently two ways we generate the necessary files. We are currently migrating the build system to use containers so the builds are reproducible. This is being done on an OS-by-OS basis. Please update this documentation as components of the build system change.

### Old Build System (currently for `GOOS != "linux"`)

The old build system generates the Go files based on the C header files present on your system. This means that files for a given GOOS/GOARCH pair must be generated on a system with that OS and architecture. This also means that the generated code can differ from system to system, based on differences in the header files.

To avoid this, if you are using the old build system, only generate the Go files on an installation with unmodified header files. It is also important to keep track of which version of the OS the files were generated from (ex. Darwin 14 vs Darwin 15). This makes it easier to track the progress of changes and have each OS upgrade correspond to a single change.

To build the files for your current OS and architecture, make sure GOOS and GOARCH are set correctly and run `mkall.sh`. This will generate the files for your specific system. Running `mkall.sh -n` shows the commands that will be run.

Requirements: bash, go

### New Build System (currently for `GOOS == "linux"`)

The new build system uses a Docker container to generate the go files directly from source checkouts of the kernel and various system libraries. This means that on any platform that supports Docker, all the files using the new build system can be generated at once, and generated files will not change based on what the person running the scripts has installed on their computer.

The OS specific files for the new build system are located in the `${GOOS}` directory, and the build is coordinated by the `${GOOS}/mkall.go` program. When the kernel or system library updates, modify the Dockerfile at `${GOOS}/Dockerfile` to checkout the new release of the source.

To build all the files under the new build system, you must be on an amd64/Linux system and have your GOOS and GOARCH set accordingly. Running `mkall.sh` will then generate all of the files for all of the GOOS/GOARCH pairs in the new build system. Running `mkall.sh -n` shows the commands that will be run.

Requirements: bash, go, docker

## Component files

This section describes the various files used in the code generation process. It also contains instructions on how to modify these files to add a new architecture/OS or to add additional syscalls, types, or constants. Note that if you are

using the new build system, the scripts/programs cannot be called normally. They must be called from within the docker container.

### asm files

The hand-written assembly file at `asm_${GOOS}_${GOARCH}.s` implements system call dispatch. There are three entry points:

```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2, err uintptr)
func Syscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2, err uintptr)
func RawSyscall(trap, a1, a2, a3 uintptr) (r1, r2, err uintptr)
```

The first and second are the standard ones; they differ only in how many arguments can be passed to the kernel. The third is for low-level use by the ForkExec wrapper. Unlike the first two, it does not call into the scheduler to let it know that a system call is running.

When porting Go to a new architecture/OS, this file must be implemented for each GOOS/GOARCH pair.

### mksysnum

Mksysnum is a Go program located at `${GOOS}/mksysnum.go` (or `mksysnum_${GOOS}.go` for the old system). This program takes in a list of header files containing the syscall number declarations and parses them to produce the corresponding list of Go numeric constants. See `zsysnum_${GOOS}_${GOARCH}.go` for the generated constants.

Adding new syscall numbers is mostly done by running the build on a sufficiently new installation of the target OS (or updating the source checkouts for the new build system). However, depending on the OS, you may need to update the parsing in mksysnum.

### mksyscall.go

The `syscall.go`, `syscall_${GOOS}.go`, `syscall_${GOOS}_${GOARCH}.go` are hand-written Go files which implement system calls (for unix, the specific OS, or the specific OS/Architecture pair respectively) that need special handling and list `//sys` comments giving prototypes for ones that can be generated.

The mksyscall.go program takes the `//sys` and `//sysnb` comments and converts them into syscalls. This requires the name of the prototype in the comment to match a syscall number in the `zsysnum_${GOOS}_${GOARCH}.go` file. The function prototype can be exported (capitalized) or not.

Adding a new syscall often just requires adding a new `//sys` function prototype with the desired arguments and a capitalized name so it is exported. However, if you want the interface to the syscall to be different, often one will make an unexported `//sys` prototype, and then write a custom wrapper in `syscall_${GOOS}.go`.

### types files

For each OS, there is a hand-written Go file at `${GOOS}/types.go` (or `types_${GOOS}.go` on the old system). This file includes standard C headers and creates Go type aliases to the corresponding C types. The file is then fed through godef to get the Go compatible definitions. Finally, the generated code is fed though mkpost.go to format the code correctly and remove any hidden or private identifiers. This cleaned-up code is written to `ztypes_${GOOS}_${GOARCH}.go`.

The hardest part about preparing this file is figuring out which headers to include and which symbols need to be `#define`d to get the actual data structures that pass through to the kernel system calls. Some C libraries preset

alternate versions for binary compatibility and translate them on the way in and out of system calls, but there is almost always a `#define` that can get the real ones. See `types_darwin.go` and `linux/types.go` for examples.

To add a new type, add in the necessary include statement at the top of the file (if it is not already there) and add in a type alias line. Note that if your type is significantly different on different architectures, you may need some `#if/#elif` macros in your include statements.

### mkerrors.sh

This script is used to generate the system's various constants. This doesn't just include the error numbers and error strings, but also the signal numbers and a wide variety of miscellaneous constants. The constants come from the list of include files in the `includes_${uname}` variable. A regex then picks out the desired `#define` statements, and generates the corresponding Go constants. The error numbers and strings are generated from `#include <errno.h>`, and the signal numbers and strings are generated from `#include <signal.h>`. All of these constants are written to `zerrors_${GOOS}_${GOARCH}.go` via a C program, `_errors.c`, which prints out all the constants.

To add a constant, add the header that includes it to the appropriate variable. Then, edit the regex (if necessary) to match the desired constant. Avoid making the regex too broad to avoid matching unintended constants.

### internal/mkmerge

This program is used to extract duplicate const, func, and type declarations from the generated architecture-specific files listed below, and merge these into a common file for each OS.

The merge is performed in the following steps:

1. Construct the set of common code that is idential in all architecture-specific files.
2. Write this common code to the merged file.
3. Remove the common code from all architecture-specific files.

## Generated files

**`zerrors_${GOOS}_${GOARCH}.go`**

A file containing all of the system's generated error numbers, error strings, signal numbers, and constants. Generated by `mkerrors.sh` (see above).

**`zsyscall_${GOOS}_${GOARCH}.go`**

A file containing all the generated syscalls for a specific GOOS and GOARCH. Generated by `mksyscall.go` (see above).

**`zsysnum_${GOOS}_${GOARCH}.go`**

A list of numeric constants for all the syscall number of the specific GOOS and GOARCH. Generated by mksysnum (see above).

**`ztypes_${GOOS}_${GOARCH}.go`**

A file containing Go types for passing into (or returning from) syscalls. Generated by godefs and the types file (see above).