This is a living document and at times it will be out of date. It is intended to articulate how programming in the Go runtime differs from writing normal Go. It focuses on pervasive concepts rather than details of particular interfaces.

## Scheduler structures

The scheduler manages three types of resources that pervade the runtime: Gs, Ms, and Ps. It's important to understand these even if you're not working on the scheduler.

### Gs, Ms, Ps

A "G" is simply a goroutine. It's represented by type `g`. When a goroutine exits, its `g` object is returned to a pool of free `g`s and can later be reused for some other goroutine.

An "M" is an OS thread that can be executing user Go code, runtime code, a system call, or be idle. It's represented by type `m`. There can be any number of Ms at a time since any number of threads may be blocked in system calls.

Finally, a "P" represents the resources required to execute user Go code, such as scheduler and memory allocator state. It's represented by type `p`. There are exactly `GOMAXPROCS` Ps. A P can be thought of like a CPU in the OS scheduler and the contents of the `p` type like per-CPU state. This is a good place to put state that needs to be sharded for efficiency, but doesn't need to be per-thread or per-goroutine.

The scheduler's job is to match up a G (the code to execute), an M (where to execute it), and a P (the rights and resources to execute it). When an M stops executing user Go code, for example by entering a system call, it returns its P to the idle P pool. In order to resume executing user Go code, for example on return from a system call, it must acquire a P from the idle pool.

All `g`, `m`, and `p` objects are heap allocated, but are never freed, so their memory remains type stable. As a result, the runtime can avoid write barriers in the depths of the scheduler.

### User stacks and system stacks

Every non-dead G has a *user stack* associated with it, which is what user Go code executes on. User stacks start small (e.g., 2K) and grow or shrink dynamically.

Every M has a *system stack* associated with it (also known as the M's "g0" stack because it's implemented as a stub G) and, on Unix platforms, a *signal stack* (also known as the M's "gsignal" stack). System and signal stacks cannot grow, but are large enough to execute runtime and cgo code (8K in a pure Go binary; system-allocated in a cgo binary).

Runtime code often temporarily switches to the system stack using `systemstack`, `mcall`, or `asmcgocall` to perform tasks that must not be preempted, that must not grow the user stack, or that switch user goroutines. Code running on the system stack is implicitly non-preemptible and the garbage collector does not scan system stacks. While running on the system stack, the current user stack is not used for execution.

### `getg()` and `getg().m.curg`

To get the current user `g`, use `getg().m.curg`.

`getg()` alone returns the current `g`, but when executing on the system or signal stacks, this will return the current M's "g0" or "gsignal", respectively. This is usually not what you want.

To determine if you're running on the user stack or the system stack, use `getg() == getg().m.curg`.

## Error handling and reporting

Errors that can reasonably be recovered from in user code should use `panic` like usual. However, there are some situations where `panic` will cause an immediate fatal error, such as when called on the system stack or when called during `mallocgc`.

Most errors in the runtime are not recoverable. For these, use `throw`, which dumps the traceback and immediately terminates the process. In general, `throw` should be passed a string constant to avoid allocating in perilous situations. By convention, additional details are printed before `throw` using `print` or `println` and the messages are prefixed with "runtime:".

For runtime error debugging, it's useful to run with `GOTRACEBACK=system` or `GOTRACEBACK=crash`.

## Synchronization

The runtime has multiple synchronization mechanisms. They differ in semantics and, in particular, in whether they interact with the goroutine scheduler or the OS scheduler.

The simplest is `mutex`, which is manipulated using `lock` and `unlock`. This should be used to protect shared structures for short periods. Blocking on a `mutex` directly blocks the M, without interacting with the Go scheduler. This means it is safe to use from the lowest levels of the runtime, but also prevents any associated G and P from being rescheduled. `rwmutex` is similar.

For one-shot notifications, use `note`, which provides `notesleep` and `notewakeup`. Unlike traditional UNIX `sleep`/`wakeup`, `note`s are race-free, so `notesleep` returns immediately if the `notewakeup` has already happened. A `note` can be reset after use with `noteclear`, which must not race with a sleep or wakeup. Like `mutex`, blocking on a `note` blocks the M. However, there are different ways to sleep on a `note`:`notesleep` also prevents rescheduling of any associated G and P, while `notetsleepg` acts like a blocking system call that allows the P to be reused to run another G. This is still less efficient than blocking the G directly since it consumes an M.

To interact directly with the goroutine scheduler, use `gopark` and `goready`. `gopark` parks the current goroutine—putting it in the "waiting" state and removing it from the scheduler's run queue—and schedules another goroutine on the current M/P. `goready` puts a parked goroutine back in the "runnable" state and adds it to the run queue.

In summary,

Blocks

Interface

G

M

P

(rw)mutex

Y

Y

Y

note

Y

Y

Y/N

park

Y

N

N

## Atomics

The runtime uses its own atomics package at `runtime/internal/atomic`. This corresponds to `sync/atomic`, but functions have different names for historical reasons and there are a few additional functions needed by the runtime.

In general, we think hard about the uses of atomics in the runtime and try to avoid unnecessary atomic operations. If access to a variable is sometimes protected by another synchronization mechanism, the already-protected accesses generally don't need to be atomic. There are several reasons for this:

1. Using non-atomic or atomic access where appropriate makes the code more self-documenting. Atomic access to a variable implies there's somewhere else that may concurrently access the variable.
2. Non-atomic access allows for automatic race detection. The runtime doesn't currently have a race detector, but it may in the future. Atomic access defeats the race detector, while non-atomic access allows the race detector to check your assumptions.
3. Non-atomic access may improve performance.

Of course, any non-atomic access to a shared variable should be documented to explain how that access is protected.

Some common patterns that mix atomic and non-atomic access are:

- Read-mostly variables where updates are protected by a lock. Within the locked region, reads do not need to be atomic, but the write does. Outside the locked region, reads need to be atomic.
- Reads that only happen during STW, where no writes can happen during STW, do not need to be atomic.

That said, the advice from the Go memory model stands: "Don't be [too] clever." The performance of the runtime matters, but its robustness matters more.

## Unmanaged memory

In general, the runtime tries to use regular heap allocation. However, in some cases the runtime must allocate objects outside of the garbage collected heap, in *unmanaged memory*. This is necessary if the objects are part of the memory manager itself or if they must be allocated in situations where the caller may not have a P.

There are three mechanisms for allocating unmanaged memory:

- sysAlloc obtains memory directly from the OS. This comes in whole multiples of the system page size, but it can be freed with sysFree.
- persistentalloc combines multiple smaller allocations into a single sysAlloc to avoid fragmentation. However, there is no way to free persistentalloced objects (hence the name).

- fixalloc is a SLAB-style allocator that allocates objects of a fixed size. fixalloced objects can be freed, but this memory can only be reused by the same fixalloc pool, so it can only be reused for objects of the same type.

In general, types that are allocated using any of these should be marked `//go:notinheap` (see below).

Objects that are allocated in unmanaged memory **must not** contain heap pointers unless the following rules are also obeyed:

1. Any pointers from unmanaged memory to the heap must be garbage collection roots. More specifically, any pointer must either be accessible through a global variable or be added as an explicit garbage collection root in `runtime.markroot`.
2. If the memory is reused, the heap pointers must be zero-initialized before they become visible as GC roots. Otherwise, the GC may observe stale heap pointers. See "Zero-initialization versus zeroing".

## Zero-initialization versus zeroing

There are two types of zeroing in the runtime, depending on whether the memory is already initialized to a type-safe state.

If memory is not in a type-safe state, meaning it potentially contains "garbage" because it was just allocated and it is being initialized for first use, then it must be *zero-initialized* using `memclrNoHeapPointers` or non-pointer writes. This does not perform write barriers.

If memory is already in a type-safe state and is simply being set to the zero value, this must be done using regular writes, `typedmemclr`, or `memclrHasPointers`. This performs write barriers.

## Runtime-only compiler directives

In addition to the "//go:" directives documented in "go doc compile", the compiler supports additional directives only in the runtime.

### go:systemstack

`go:systemstack` indicates that a function must run on the system stack. This is checked dynamically by a special function prologue.

### go:nowritebarrier

`go:nowritebarrier` directs the compiler to emit an error if the following function contains any write barriers. (It *does not* suppress the generation of write barriers; it is simply an assertion.)

Usually you want `go:nowritebarrierrec`. `go:nowritebarrier` is primarily useful in situations where it's "nice" not to have write barriers, but not required for correctness.

## go:nowritebarrierrec and go:yeswritebarrierrec

`go:nowritebarrierrec` directs the compiler to emit an error if the following function or any function it calls recursively, up to a `go:yeswritebarrierrec`, contains a write barrier.

Logically, the compiler floods the call graph starting from each `go:nowritebarrierrec` function and produces an error if it encounters a function containing a write barrier. This flood stops at `go:yeswritebarrierrec` functions.

`go:nowritebarrierrec` is used in the implementation of the write barrier to prevent infinite loops.

Both directives are used in the scheduler. The write barrier requires an active P (`getg().m.p != nil`) and scheduler code often runs without an active P. In this case, `go:nowritebarrierrec` is used on functions that release the P or may run without a P and `go:yeswritebarrierrec` is used when code re-acquires an active P. Since these are function-level annotations, code that releases or acquires a P may need to be split across two functions.

## go:notinheap

`go:notinheap` applies to type declarations. It indicates that a type must never be allocated from the GC'd heap or on the stack. Specifically, pointers to this type must always fail the `runtime.inheap` check. The type may be used for global variables, or for objects in unmanaged memory (e.g., allocated with `sysAlloc`, `persistentalloc`, `fixalloc`, or from a manually-managed span). Specifically:

1. `new(T)`, `make([]T)`, `append([]T, ...)` and implicit heap allocation of T are disallowed. (Though implicit allocations are disallowed in the runtime anyway.)
2. A pointer to a regular type (other than `unsafe.Pointer`) cannot be converted to a pointer to a `go:notinheap` type, even if they have the same underlying type.
3. Any type that contains a `go:notinheap` type is itself `go:notinheap`. Structs and arrays are `go:notinheap` if their elements are. Maps and channels of `go:notinheap` types are disallowed. To keep things explicit, any type declaration where the type is implicitly `go:notinheap` must be explicitly marked `go:notinheap` as well.
4. Write barriers on pointers to `go:notinheap` types can be omitted.

The last point is the real benefit of `go:notinheap`. The runtime uses it for low-level internal structures to avoid memory barriers in the scheduler and the memory allocator where they are illegal or simply inefficient. This mechanism is reasonably safe and does not compromise the readability of the runtime.