Notes on the Go 1.4 Run-Time

Francisco J. Ballesteros
TR LSUB-15-2 (rev 1)

ABSTRACT

This TR contains notes about the Go run time as of Go 1.4. They are intended to aid in the construction of a kernel for Clive. The description here will be updated in the future without publising a new TR, the TR revision number can be used to see if the version is up to date or not.

1. Initialization

- `rt0_darwin_amd64.s:13 main()` is the entry point for the binary. A direct jump to...

- `asm_amd64.s:9: runtime.rt0_go` which is the actual crt0. This initializes the stack calls `_cgo_init` if needed, updates stack guards, setups TLS, sets `m` to `m0` and `g` to `g0` and continues calling...

  - `runtime.args` to save the arguments from the OS
  - `os_darwin.c:54: runtime.osinit` to initialize threading (they don't use their libc)
  - `proc.c:122: runtime.schedinit(void)` to create a G that calls `runtime.main`.
    - `traceback.go:48: func tracebackinit()`
    - `symtab.go:46: func symtabinit`
    - `stack.c:42: runtime.stackinit(void)` calls this function for the stacks in the pool:
      - `mheap.c:663: runtime.MSpanList_Init`
    - `malloc.c:109: runtime.mallocinit`
      - `mheap.c:57: runtime.MHeap_Init` initializes the heap using these:
        - `mfixalloc.c:16: runtime.FixAlloc_Init`, for several allocs.
        - `mheap.c:663: runtime.MSpanList_Init`, for free and busy lists.
        - `mcentral.c:21: runtime.MCentral_Init`, for mcentral lists.
      - `mcache.c:19: runtime.allocmcache` sets `g->m->mcache`, a per-P malloc cache for small objects.
    - `proc.c:190: mcommoninit` calls os-specific `mpreinit` and links m into the sched list (`allm`).
    - `goargs` and `goenvs` save the UNIX arguments.
    - `mgc0.c:1345: runtime.gcinit` sets GC masks for data and bss.
    - `proc.c:2655: procresize` is called to create *Ps* for `GOMAXPROCS`.
  - `proc.c:2154: runtime.newproc` is called to run `runtime.main` using

- `asm_amd64.s:218: runtime.onM` to call `newproc_m)` to run `runtime.main`. This switches to g0, if not on m stack (i.e., no switch this time).
  - `proc.c:2128: newproc_m` is called by g0 and copies the args and then calls:
    - `proc.c:2182: runtime.newproc1`. This creates a G to run a function (`runtime.main`). Using:
      - `proc.c:2295: gfget` to get a G from the free list, or
      - `proc.c:2102: runtime.malg` to allocate a G, which ends up doing a `new(g)` in go.

      Then it copies arguments to the new G stack, sets the sched label to call the function, using
      - `stack.c:806: runtime.gostartcallfn`, similar to setlabel.

      Sets the state to Grunnable and calls
      - `proc.c:3284: runqput` to put the new G on a local runq or the global one.
      - `proc.c:1276: wakep`, which is a nop or a call to sched the M for P or make a new M for P.
        - `proc.c:1200: startm`
- `proc.c:818: runtime.mstart` starts the M. It calls:
  - `asminit` to do per-thread initialization (nop for our arch).
  - `os_darwin.c:144: minit` initializes signal handling for our arch.
  - `proc.c:1537: schedule` runs the scheduller and calls this when gets something to run:
    - `proc.c:1358: execute` is mostly a call to a goto-label:
      - `asm_amd64.s:143: gogo`

At the end, the new G calls

- `proc.go:16: runtime.main()`, which calls
  - `runtime_init`
  - `main_init`
  - `main_main`
  - `exit(0)`

These are the initialization code and main program for the binary. The `runtime.init` function in `proc.go` spawns the goroutine to call `gogc(1)` in a loop.

2. Labels

A `Gobuf` is similar to a *Label*:

```
struct    Gobuf
{
    uintptr    sp;    // the actual label.
    uintptr    pc;    // the actual label.
    G*    g;    // the actual label.
    void*    ctxt;    // aux context pointer.
    uintreg    ret;    // return value
    uintptr    lr;    // link register used in ARM.
};
```

The main operations are:

- `runtime.gosave`, i.e., set label. Clears `ctxt` and `ret`.
- `runtime.gogo`, i.e., goto label. Sets `ctxt` in `DX`, used by `runtime.morestack` and returns `ret` after the actual jump to the label.

Here, `pc`, `sp`, and `g` are the actual context saved and restored. That is the label.

The field `ctxt` is used to point to a frame in the stack and seems to be the user context in other places. Seems to be an auxiliary pointer but it is not clear (yet) how it is used. The field `lr` seems to be the link register for the ARM and is not used by our architecture.

3. Sleep/Wakeup

These are giant locks:

```
void    runtime.stoptheworld(void);
void    runtime.starttheworld(void);
extern uint32 runtime.worldsema;
```

used for example to dump all the stacks.

These are the usual locks with a user-level fast path:

```
void    runtime.lock(Mutex*);
void    runtime.unlock(Mutex*);
```

These are sleep/wakeup like structures, and timed out variants for them with the usual conventions of at most one calling sleep and at most one calling wakeup for it. Plus, there is a clear operation to reset the thing for a further use:

```
void    runtime.noteclear(Note*);
void    runtime.notesleep(Note*);
void    runtime.notewakeup(Note*);
bool    runtime.notetsleep(Note*, int64);  // false - timeout
bool    runtime.notetsleepg(Note*, int64);  // false - timeout
```

And these are futexes or semaphores (eg., on Darwin) to implement the ones above:

```
uintptr    runtime.semacreate(void);
int32    runtime.semasleep(int64);
void    runtime.semawakeup(M*);
```

In many cases lock-free data structures are used; or at least parts of them are handled lock-free by using atomic and CAS-like operations, eg. to change statuses of processes and to link them to a list.

4. Scheduling

Most of the interesting code is in `proc.c`. There are three central structures:

- *G*

  A goroutine. `g` refers to the current and `g0` is the idle process. This is a user-level thread. `g` is a register on the ARM and a slot in TLS everywhere else.
- *M*

  A machine, actually a UNIX process. `m` refers to the current one. This is a kernel-level thread that runs *Gs* or is idle or is in a syscall.
- *P*

  A processor, actually a per `GOMAXPROCS` sched.

The idea is that *M* takes a *P* when it must run *Gs*. *Ps* were introduced to do job stealing.

The interesting bits of *G* are:

```
struct    G
{
    Stack    stack;    // [stack.lo, stack.hi).
    uintptr    stackguard0;    // stack.lo+StackGuard or StackPreempt
    uintptr    stackguard1;    // stack.lo+StackGuard on g0 or ˜0 on others

    Panic*    panic;    // innermost panic - offset known to liblink
    Defer*    defer;    // innermost defer
    Gobuf    sched;
    uintptr    syscallsp;    // if status==Gsyscall, syscallsp = sched.sp to use during gc
    uintptr    syscallpc;    // if status==Gsyscall, syscallpc = sched.pc to use during gc
    void*    param;    // passed parameter on wakeup
    int64    goid;
    G*    schedlink;
    bool    preempt;    // preemption signal, dup of stackguard0 = StackPreempt
    M*    m;    // for debuggers, but offset not hard-coded
    M*    lockedm;
    SudoG*    waiting;    // sudog structures this G is waiting on
    ...
};
```

The interesting bits of *M* are:

```
struct    M
{
    G*    g0;        // goroutine with scheduling stack
    G*    gsignal;    // signal-handling G
    uintptr    tls[4];        // thread-local storage (for x86 extern register)
    void    (*mstartfn)(void);
    G*    curg;        // current running goroutine
    G*    caughtsig;    // goroutine running during fatal signal
    P*    p;        // attached P for executing Go code (nil if not executing Go code)
    int32    mallocing,throwing,gcing;
    int32    locks;
    bool    spinning;    // M is out of work and is actively looking for work
    bool    blocked;    // M is blocked on a Note
    Note    park;
    M*    alllink;    // on allm
    M*    schedlink;
    MCache*    mcache;
    G*    lockedg;
    uint32    locked;        // tracking for LockOSThread
    M*    nextwaitm;    // next M waiting for lock
    uintptr    waitsema;    // semaphore for parking on locks
    uint32    waitsemacount;
    uint32    waitsemalock;
    bool    (*waitunlockf)(G*, void*);
    void*    waitlock;
    uintptr scalararg[4];    // scalar argument/return for mcall
    void*   ptrarg[4];    // pointer argument/return for mcall
    ...
};
```

A value of `m->locks` greater than zero prevents preemption and also prevents garbage colloection.

The interesting bits of *P* are:

```
struct P
{
    Mutex    lock;
    uint32    status;        // Pidle, Prunning, Psyscall, Pgcstop, Pdead
    P*    link;
    uint32    schedtick;        // incremented on every scheduler call
    uint32    syscalltick;    // incremented on every system call
    M*    m;        // back-link to associated M (nil if idle)
    MCache*    mcache;
    Defer*    deferpool[5];    // pool of available Defers

    // Cache of goroutine ids, amortizes accesses to runtime.sched.goidgen.
    uint64    goidcache;
    uint64    goidcacheend;

    // Queue of runnable goroutines.
    uint32    runqhead;
    uint32    runqtail;
    G*    runq[256];

    // Available G's (status == Gdead)
    G*    gfree;
    int32    gfreecnt;
};
```

This is a local scheduler plus cached structures per UNIX process used to run Go code. And then there is a global scheduler structure that also caches some structures.

```
struct    SchedT
{
    Mutex    lock;
    uint64    goidgen;
    M*    midle;     // idle m's waiting for work
    P*    pidle;      // idle P's
    G*    runqhead; // Global runnable queue.
    Mutex    gflock;    // global cache go dead Gs
    G*    gfree;
    uint32    gcwaiting;    // gc is waiting to run
    int32    stopwait;
    Note    stopnote;
    ...
};
```

There is a single `runtime.sched`, and `runtime.sched.lock` protects it all.

4.1. Process creation

Code like `go fn()` is translated as a call to `proc.c:2154: runtime.newproc` as shown in the first section for initialization. This records the function and a pointer to the `...` arguments yn `g->m->ptrarg[]` and then calls `newproc_m` using `runtime.onM`.

- There's a switch to `M`'s `g0`
- `newproc_m` runs there.
- And there's a switch back when done.

Then `newproc_m` takes the function and arguments from `m->ptrarg` and calls `runtime.newproc1`:

- This adds to `m->locks` to avoid preemption and takes *P* from `m->p`.
- A *G* is taken from the cache at *P* or allocated (with `StackMin` stack bytes).

At this point *G* is `Gdead`. Then arguments are copied into *G*'s stack and it's prepared:

- It's label `pc`, `sp`, and `g` are set to the function pointer and the new stack and `g`, and then adjusted to pretend that such function did call `gosave`, so we can `gogo` (gotolabel) it. In this case, the label `ctxt` is set to the `FuncVal` value going. The return value is set to call `goexit+PCQuantum` which is a call to `goexit1`.
- Its state is changed to `Grunnable` (cas op.)
- A new `id` is taken from the id cache at `P` (perhaps refilling the cache).

Now it's put in the scheduler queue by calling `runqput(p, newg)`. This uses atomic load/store to put `newg` at `p->runq[tail]`, using `p->runqtail` as a increasing counter for the tail that is copied to `tail` and truncated as an index for `runq`. This happens if `p->runq` is not full.

If `p->runq` is full, `runqputslow` is called to take half of the local queue at *P* and place them at the global `runtime.sched.runqhead/tail` queue while holding the global scheduler lock. if the CAS operation to operate on the local queue fails, the whole `runqput` is retried.

4.2. Process termination

Performed by a call to

- `proc.c:1682: runtime.goexit1`, that does an `mcall` to run `goexit0` at `g0`.
- `goexit0` (or any other mcall) never returns and calls `gogo` to sched to somewhere else, usually to `g->sched` to let it run later. The *G* state is set to `Gdead`. Then these are called:
  - `proc.c:1598: dropg`, to release `m->curg->m` and `m->curg` (`g` is now `g0`).
  - `proc.c:2259: gfput`, to put the dying *G* at `p->gfree`, linked through `g->schedlink`. If there are too many free *G*s cached, they are moved to the global list at `runtime.sched.gfree`.
  - `schedule`, runs one scheduler loop and jumps to a new *G*.

Note that returning from the main function of a goroutine returns to

- `asm_amd64.s:2237: runtime.goexit`, which calls `runtime.goexit1`.

4.3. Context switches

Calls to `schedule` can be found at:

- `proc.c:661: mquiesce`, used to run code at `g0` and then schedule back a *G*.
- `proc.c:868: mstart`, to jump to a new `g0` for a new *M*.
- `proc.c:1655: runtime.park_m`, used to make *G* wait for something.
- `proc.c:1675: runtime.gosched_m`, an `mcall` from `Gosched` at `proc.go`.
- `proc.c:1718: goexit0`, to run `g0` or other *G*s when exiting.
- `proc.c:2022: exitsyscall0`, called after a system call at `g0`.
- `malloc.go:482: gogc` calls `Gosched` when the GC is done.
- `mgc0.go:87: bgsweep` calls `Gosched`.

Plus, if `g->stackguard0` is `StackPreempt`, then `asm_amd64.s:824:runtime.stackcheck` calls `morestack` and preemts if the goroutine seems to be running user code (not runtime code), by calling `gosched_m` like `Gosched` does. I have not checked out if the compiler inserts calls in loops that do not perform function calls (which check the stack), and those might never be preempted; which does not matter much for us now becase they would be bugs in the code and not the normal case. The stack checks are inserted silently by the linker unless `NOSPLIT` is indicated, and thus normal function calls are a source of possible preemptions.

As a side-efect, a `NOSPLIT` function call is not preemptible by this mechanism.

The code in `schedule` runs the scheduler:

- There can be no `m->locks`, or it's a bug.
- If `m->lockedg` then
  - `proc.c:1287: stoplockedm` is called, to stop the current *G* at *M* until *G* can run again. This is done by lending `m->p` to another *M*, if we have a *P* using `handoffp(releasep())`, and then calling `runtime.notesleep(&g->m->park)` to sleep. Later `acquirep(m->nextp)` is used to get a new *P* before returning and...
  - `proc.c:1358: execute` is called for `m->lockedg`.

- If `runtime.sched.gcwaiting` then `gcstopm` is called, which calls `releasep` and `stopm`; This happens when `stoptheworld` is called and, once we are done `schedule` restarts again.

At this point `schedule` picks a ready *G*. Once in a while from the global pool, and usually from `m->p`.

- `proc.c:3333: runqget` picks one *G* from the queue using `p->runqhead` and CAS.
  - When there is no *G* ready to run, `proc.c:1381: findrunnable` tries to steal calling `globrunqget` and, `runtime.netpoll` (!!).
  - Then if there's no work, `m->spinning` is set and it tries to steal from others.
  - When everything fails, it calls `stopm` and blocks.

Once it gets a *G* to run:

- `proc.c:1358: execute` is called, which is mostly a call to a goto-label:
  - `asm_amd64.s:143: gogo`

5. System calls

All system calls call

- `asm_darwin_amd64.s:19 syscall.Syscall` or `Syscall6` that
  - calls `proc.c:1761 runtime.reentersyscall` (from `runtime.entersyscall`) to prepare for a system call
    - This increments `m->locks` to avoid preemption, and tricks `g->stackguard0` to make sure the no split-stack happens.
    - Sets *G* status to `Gsyscall`
    - Saves the PC and SP in `g->sched`.
    - Releases `p->m` and `m->mcache`
    - Sets *P* status to `Psyscall`
    - Calls `entersyscall_gcwait` when `runtime.sched.gcwaiting`

  - Decrements `m->locks` before proceding further.
- moves parameters into registers and executes
- executes `SYSCALL`, and, upon return from the system call, calls
- calls `proc.c:1893 runtime.exitsyscall` calls directly
  - `exitsyscallfast` to either re-acquire the last *P* or or get another idle *P*.
  - or runs `exitsyscall0` as an `mcall` to run the scheduler and then return to continue when the scheduler returns and *G* can run again.
    - This calls `pidleget` and then `acquirep` and `execute` or `stopm` and `schedule`, depending.
- and returns to the caller.

There are other wrappers for system calls but in the end they call the above entry points.

Two other functions, `syscall.RawSyscall` and `.RawSyscall6` are wrappers that issue the system call without doing anything: no calls to `runtime.entersyscall` and no calls to `runtime.exitsyscall`. I don't know why they are not called from the entry points above.

6. Signals

There is a global `runtime.sigtab[sig]` table with entries of this data type:

```
struct    SigTab
{
    int32    flags;
    int8    *name;
};
enum
{
    SigNotify = 1<<0,    // let signal.Notify have signal, even if from kernel
    SigKill = 1<<1,        // if signal.Notify doesn't take it, exit quietly
    SigThrow = 1<<2,    // if signal.Notify doesn't take it, exit loudly
    SigPanic = 1<<3,    // if the signal is from the kernel, panic
    SigDefault = 1<<4,    // if the signal isn't explicitly requested, don't monitor it
    SigHandling = 1<<5,    // our signal handler is registered
    SigIgnored = 1<<6,    // the signal was ignored before we registered for it
    SigGoExit = 1<<7,    // cause all runtime procs to exit (only used on Plan 9).
};
```

Signals are handled in their own signal stack. When a new *M* is initialized,

- `runtime.minit` is called, eg., for Darwin, it calls
  - `sys_darwin_amd64.s:265: runtime.sigaltstack` to set the signal stack context, and
  - `sys_darwin_386.s:218: runtime.sigprocmask` to set the signal mask to none.

Later on, when `dropm` is called to release an *M*,

- `runtime.unminit` is called and it calls `runtime.signalstack` to unset the signal stack.

The `signal` package `init` function:

- calls `runtime.signal_enable`, which when called for the first time sets `sig.inuse` and clears the `sig.note` note, to prepare for handling signals.

- loops calling `process(syscall.Signal(signal_recv()))`. Here,
  - `sig.s:21 signal_recv` is just a jump to
  - `sigqueue.go:91 runtime.signal_recv`, which returns a signal number when it's posted by the OS.

When `signal.Notify` is called to install a handler:

- `signal.go:49 Notify` records in a `handlers` table for the signal given or for all the signals, and calls
  - `signal_unix.go:47 signal.enableSignal`, which is just a call to
  - `sigqueue.go:128 runtime.signal_enable`. This sets the signal in `sig.wanted` and calls `sigenable_go` which is just a call, using `onM` to this:
    - `signal.c:8 runtime.sigenable_m` enables a signal calling with the signal number kept at `m->scalararg[0]`. It is just a call to:
    - `signal_unix.c:44 runtime.sigenable` enables a signal, making a call to the next with `runtime.sighandler` as the handler function.
      - `os_darwin.c:519: runtime.setsig` calls `runtime.sigaction` specifying `runtime.sigtramp` as the handler and supplying the handler function (runtime.sighandler) in the *sigaction* structure.

Later, when a signal arrives:

- `sys_darwin_386.s:240 runtime.sigtramp`
  - saves `g` and sets `m->gsignal` as the current *G*, and then calls the actual handler:
    - `signal_amd64x.c:44 runtime.sighandler` looks into the global `runtime.sigtab` and might panic if not handled or deliver the signal by calling the next.
      - `sigqueue.go:48 runtime.sigsend` is called with a signal number to send the signal to receivers. The signal is queued in a global if there is no receiver ready but the signal is wanted and is not already in the queue.

From here, the loop started in the `singal.init` function would send the signal to the users channel with a non-blocking send. That is, if there is not enough buffering in the channel signals are lost; but that's ok.

7. Memory allocation

Memory is allocated by calls to `new`. The implementation for this builtin is:

- `malloc.go:348 newobject`, which calls to `mallocgc` using `flagNoScan` as a flag if the type is marked as having no pointers.
  - `malloc.go:46 mallocgc` is the main entry point for memory allocation.

Slices are created by calling `malloc.go:357 newarray`, which does the same. Raw memory is allocated by calls to `rawmem` which also calls `mallocgc` with flags `flagNoScan` and `flagNoZero`; i.e., that's almost a direct `malloc` call, but for the GC.

There are several allocation classes. Sizes are recorded in `runtime.class_to_size[]`. Alignments are:

- 8, for sizes under 16 bytes and sizes that are not a power of 2.
- 16, for sizes under 128 bytes,
- size/8, for sizes under 2KiB,
- 256, for sizes starting at 2KiB.

Regarding sizes:

- Sizes under `MaxSmallSize` (32KiB), use the larger size that keeps the same number of objects in each page. They are allocated from a per-*P* cache.
- Sizes starting at 32KiB are rounded to `PageSize`. They are allocated from the global heap.

The allocator used depends on the allocation:

- Objects with no pointers and less than 16 bytes (`maxTinySize`) are allocated within a single allocation and share it. This is done in the *P* cache. The allocation is released when all such objects are collected. This is done in the *P* cache.
- Objects up to 1024-8 bytes go into a size class rounded to multiples of 8 bytes.
- Objects up to 32KiB are rounded to multiples of 128 bytes. This is done in the *P* cache.
- Objects larger than 32KiB are allocated on the heap by calling `largeAlloc_m` `onM`.

The heap operates using pages, but smaller allocators up in the hierarchy uses bytes. This is how it works:

- The tiny allocations use the `tiny` allocation from the `MCache` structure at `p->mcache`. There is one per *P* and it requires no locks:

  ```
  struct MCache
  {
      byte*    tiny;
      uintptr    tinysize;
      // The rest is not accessed on every malloc.
      MSpan*    alloc[NumSizeClasses];    // spans to allocate from

      StackFreeList stackcache[NumStackOrders];
      SudoG*    sudogcache;
      ...
  };
  ```

  When `tiny` is exhausted, a new one is taken from `Mcache.alloc`, and, if that is exhausted, a new one is allocated by a call to `runtime.MCache_Refill`, using `onM`. This asks `runtime.MCentral_CacheSpan` to allocate a new span and places it into `Mache.alloc`.

- The allocations with pointers or starting at 16 bytes, try first to use `p->cache.alloc` to get an allocation of the right size. If this fails, `MCache_Refill` is called as before, `onM`.

- Large allocations starting at 32KiB call `largeAlloc_m` `onM`.
  - `malloc.c:372: runtime.largeAlloc_m` rounds the size to a multiple of `PageSize` and calls:
    - `mheap.c:231: runtime.MHeap_Alloc` calls `mheap_alloc` directly if it's a call made by `g0`, or `mcall`s it.
      - `mheap.c:171 mheap_alloc` allocates n memory pages.
    - `mgc0.c:1815: runtime.markspan` is called to tell the GC.

Unless the allocation is `flagNoScan`, the GC bitmap is updated by looking at the type given to `mallocgc` to record which words are pointers. Also, the unused part at the end of the actual allocation is marked in the GC bitmap as dead.

Looking at the `MHeap` now, for allocations under 1Mbyte (for 4KiB pages), there is a list of allocation with exactly that page size. For larger allocations there is a final large list used.