Coding for Concurrency

Processes, Threads, and Goroutines

Nicholas M. Boers

Edmonton Go

November 28, 2016

Outline

Coding for concurrency

Concurrency vs. parallelism

In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about dealing with lots of things at once.

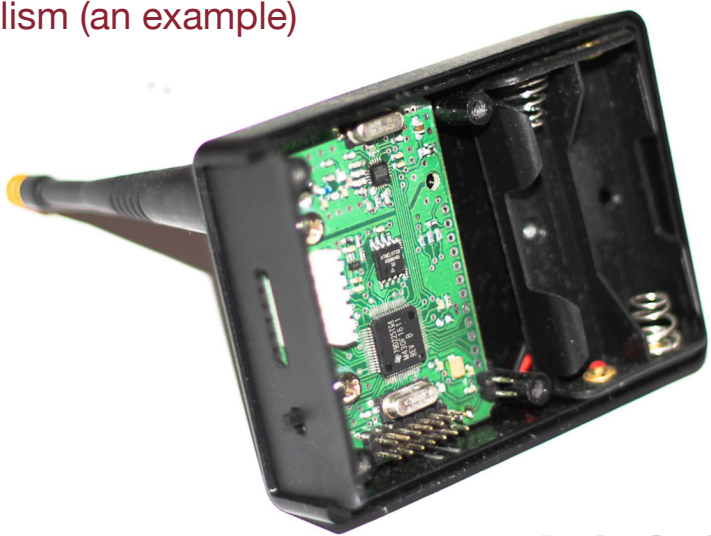
Parallelism is about doing lots of things at once.

Source: `https://blog.golang.org/concurrency-is-not-parallelism`

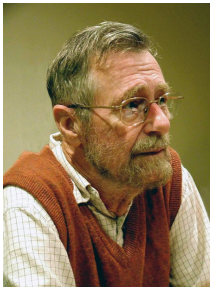
Concurrency $\neq$ parallelism (an example)

MSP430F1611

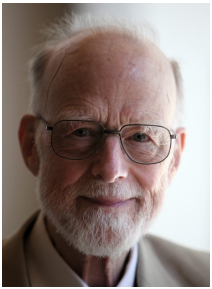
- single-core
- $\leq$ 8 MHz
- 48 KB Flash
- 10 KB RAM



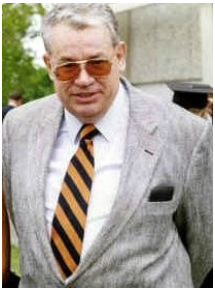
Founding fathers



Edsger W. Dijkstra



Tony Hoare



Per Brinch Hansen

Sources: P. B. Hansen, "The invention of concurrent programming," in *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, P. B. Hansen, Ed. New York, NY: Springer New York, 2002, pp. 3–61, ISBN: 978-1-4757-3472-0. DOI: 10.1007/978-1-4757-3472-0_1. [Online]. Available: http://dx.doi.org/10.1007/978-1-4757-3472-0_1; Edsger W. Dijkstra [photo by Hamilton Richards, CC BY-SA 3.0]; Tony Hoare [photo by Rama, Wikimedia Commons, CC BY-SA 2.0 fr]; Per Brinch Hansen [attribution].

Concurrency and Go

Go has been designed for concurrency:

- goroutines
- channels
- `select`

Our Go programs don't run in isolation…

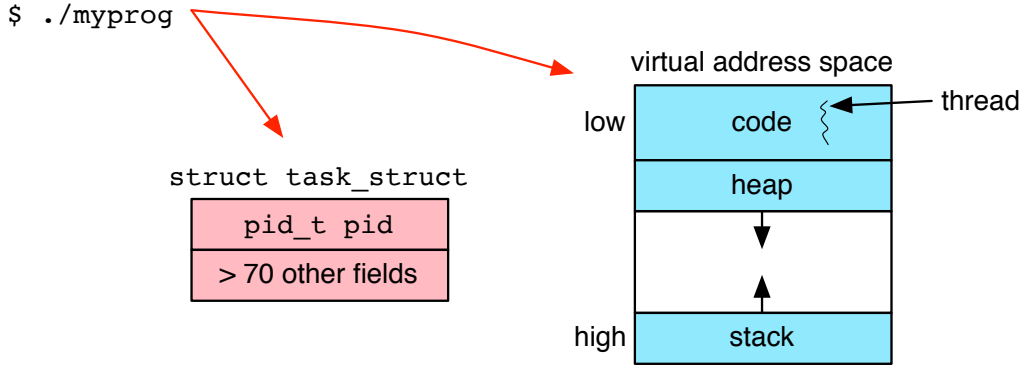
Roadmap

Let's first focus on a binary running within a Linux environment…

- processes
- threads

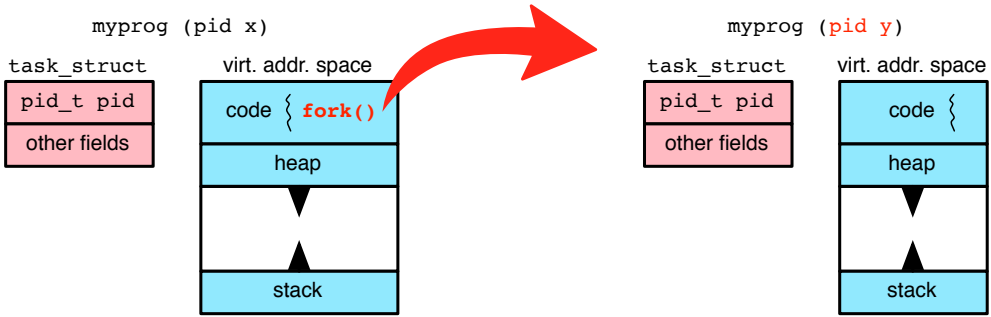
I'll mention system calls (and library functions) from the C API.

Processes



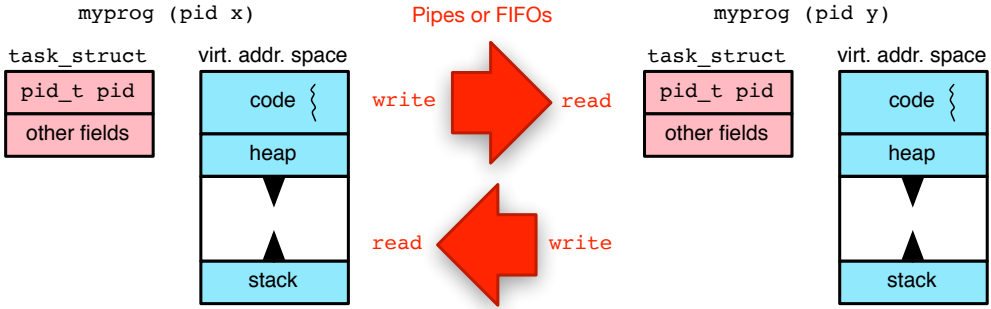
The file `include/linux/sched.h` defines `struct task_struct`.

Process → `fork` → concurrent processes (1003.1-1988)



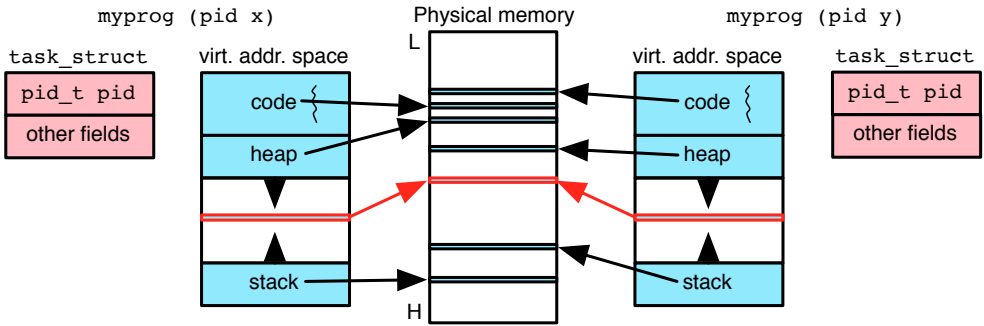
Relevant function: `fork`

Interprocess communication (IPC): Pipes or FIFOs



Some relevant functions: `pipe`, `mknod`, `open`, `read`, `write`

Interprocess communication (IPC): Shared memory (1003.1b-1993)



Some relevant functions: `shmget`, `shmat`, `shmdt`, `shmctl`

Interprocess communication (IPC): Other approaches

Some additional IPC possibilities:

- signals
- file locking
- semaphores
- memory-mapped files
  - file locking & semaphores
- UNIX sockets

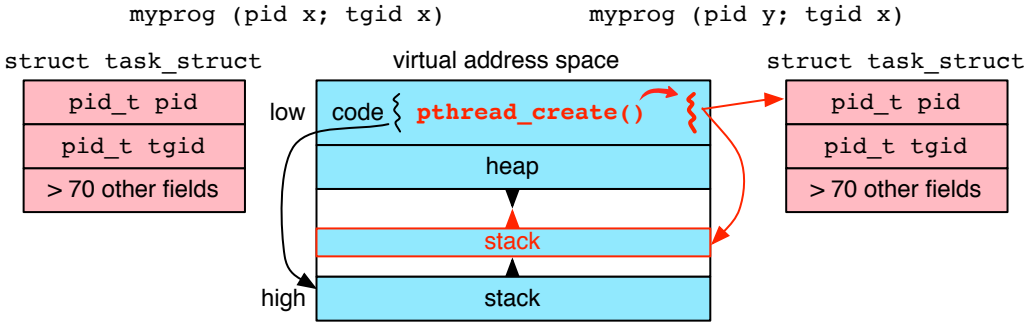
Processes: Summary

Concurrency with processes:

- various approaches to IPC
  - synchronization
- considerable overhead
  - kernel data structure(s)
  - virtual address space

When we execute a Go binary, a single process is created (in a sense).

Thread → `pthread_create` → concurrent threads (1003.1c-1995)



Many relevant functions: next slide…

POSIX Threads API

The POSIX Threads API contains a few functions:

```
pthread_atfork (3p)
pthread_attr_destroy (3p)
pthread_attr_getdetachstate (3p)
pthread_attr_getguardsize (3p)
pthread_attr_getinheritsched (3p)
pthread_attr_getschedparam (3p)
pthread_attr_getschedpolicy (3p)
pthread_attr_getscope (3p)
pthread_attr_getstack (3p)
pthread_attr_getstackaddr (3p)
pthread_attr_getstacksize (3p)
pthread_attr_init (3p)
pthread_attr_setdetachstate (3p)
pthread_attr_setguardsize (3p)
pthread_attr_setinheritsched (3p)
pthread_attr_setschedparam (3p)
pthread_attr_setschedpolicy (3p)
pthread_attr_setscope (3p)
pthread_attr_setstack (3p)
pthread_attr_setstackaddr (3p)
pthread_attr_setstacksize (3p)
pthread_barrier_destroy (3p)
pthread_barrier_init (3p)
pthread_barrier_wait (3p)
pthread_barrierattr_destroy (3p)
pthread_barrierattr_getpshared (3p)
pthread_barrierattr_init (3p)
pthread_barrierattr_setpshared (3p)
pthread_cancel (3p)
pthread_cleanup_pop (3p)
pthread_cleanup_push (3p)
pthread_cond_broadcast (3p)
pthread_cond_destroy (3p)
pthread_cond_init (3p)
pthread_cond_signal (3p)
pthread_cond_timedwait (3p)
pthread_cond_wait (3p)
pthread_condattr_destroy (3p)
pthread_condattr_getclock (3p)
pthread_condattr_getpshared (3p)
pthread_condattr_init (3p)
pthread_condattr_setclock (3p)
pthread_condattr_setpshared (3p)
pthread_create (3p)
pthread_detach (3p)
pthread_equal (3p)
pthread_exit (3p)
pthread_getconcurrency (3p)
pthread_getcpuclockid (3p)
pthread_getschedparam (3p)
pthread_getspecific (3p)
pthread_join (3p)
pthread_key_create (3p)
pthread_key_delete (3p)
pthread_kill (3p)
pthread_mutex_destroy (3p)
pthread_mutex_getprioceiling (3p)
pthread_mutex_init (3p)
pthread_mutex_lock (3p)
pthread_mutex_setprioceiling (3p)
pthread_mutex_timedlock (3p)
pthread_mutex_trylock (3p)
pthread_mutex_unlock (3p)
pthread_mutexattr_destroy (3p)
pthread_mutexattr_getprioceiling (3p)
pthread_mutexattr_getprotocol (3p)
pthread_mutexattr_getpshared (3p)
pthread_mutexattr_gettype (3p)
pthread_mutexattr_init (3p)
pthread_mutexattr_setprioceiling (3p)
pthread_mutexattr_setprotocol (3p)
pthread_mutexattr_setpshared (3p)
pthread_mutexattr_settype (3p)
pthread_once (3p)
pthread_rwlock_destroy (3p)
pthread_rwlock_init (3p)
pthread_rwlock_rdlock (3p)
pthread_rwlock_timedrdlock (3p)
pthread_rwlock_timedwrlock (3p)
pthread_rwlock_tryrdlock (3p)
pthread_rwlock_trywrlock (3p)
pthread_rwlock_unlock (3p)
pthread_rwlock_wrlock (3p)
pthread_rwlockattr_destroy (3p)
pthread_rwlockattr_getpshared (3p)
pthread_rwlockattr_init (3p)
pthread_rwlockattr_setpshared (3p)
pthread_self (3p)
pthread_setcancelstate (3p)
pthread_setcanceltype (3p)
pthread_setconcurrency (3p)
pthread_setschedparam (3p)
pthread_setschedprio (3p)
pthread_setspecific (3p)
pthread_sigmask (3p)
pthread_spin_destroy (3p)
pthread_spin_init (3p)
pthread_spin_lock (3p)
pthread_spin_trylock (3p)
pthread_spin_unlock (3p)
pthread_testcancel (3p)
```

Inter-thread communication

Unlike with separate processes, threads within the same process share a virtual address space.

- challenge: synchronization

Threads: Summary

Concurrency with threads:

- synchronization can be complicated
- (still) considerable overhead
  - kernel data structure(s)

When we execute a Go binary, several threads are created.

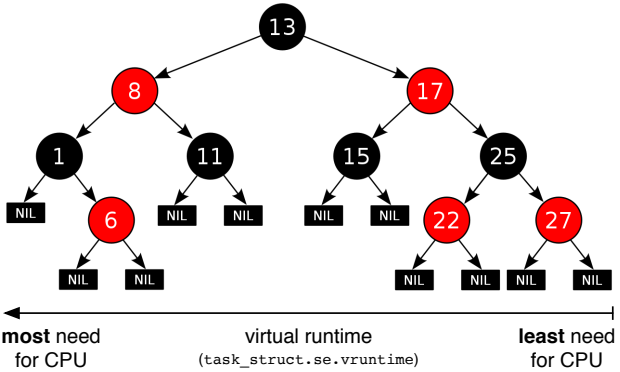
Scheduler

Linux kernel schedules each *process* separately.

- initial thread $\rightarrow$ PCB
- each additional thread $\rightarrow$ PCB

For each CPU core, it maintains a run queue implemented as a red-black tree.

Completely fair scheduler (CFS)



Sources: `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`; adapted from image by Cburnett, CC BY-SA 3.0; J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler: A decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: ACM, 2016, 1:1–1:16, ISBN: 978-1-4503-4240-7. DOI: `10.1145/2901318.2901326`. [Online]. Available: `http://doi.acm.org/10.1145/2901318.2901326`

Scheduling: Summary

Scheduling processes (including threads):

- considerable costs exist when changing processes/threads, e.g.,
  - saving registers (in `struct task_struct`)
  - restoring registers (from `struct task_struct`)
  - cache invalidation

Go routines

As already highlighted, Go provides

- goroutines
- channels
- `select`

The *operating system* scheduler is oblivious to Go routines.

How are they scheduled, then?

Details on the scheduler

The Go runtime schedules goroutines.

Details were harder to come by than expected:

- `https://golang.org/src/runtime/proc.go` refers to `https://golang.org/s/go11sched`, which is surprisingly rough
- `http://morsmachine.dk/go-scheduler` describes the design document
- `https://golang.org/doc/go1.2`
  - "1.2 is a smaller delta than the step from 1.0 to 1.1, but it still has some significant developments, including a **better scheduler**"
- `https://golang.org/doc/go1.5`
  - "the order in which goroutines are scheduled has been changed"

Details on the scheduler

The Go runtime schedules goroutines.

Details were harder to come by than expected:

- `https://golang.org/src/runtime/proc.go` refers to `https://golang.org/s/go11sched`, which is surprisingly rough
- `http://morsmachine.dk/go-scheduler` describes the design document
- `https://golang.org/doc/go1.2`
  - "1.2 is a smaller delta than the step from 1.0 to 1.1, but it still has some significant developments, including a **better scheduler**"
- `https://golang.org/doc/go1.5`
  - "the order in which goroutines are scheduled has been changed"

Details on the scheduler

The Go runtime schedules goroutines.

Details were harder to come by than expected:

- `https://golang.org/src/runtime/proc.go` refers to `https://golang.org/s/go11sched`, which is surprisingly rough
- `http://morsmachine.dk/go-scheduler` describes the design document
- `https://golang.org/doc/go1.2`
  - "1.2 is a smaller delta than the step from 1.0 to 1.1, but it still has some significant developments, including a **better scheduler**"
- `https://golang.org/doc/go1.5`
  - "the order in which goroutines are scheduled has been changed"

Details on the scheduler

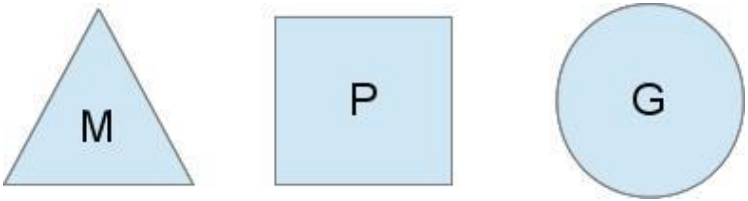
The Go runtime schedules goroutines.

Details were harder to come by than expected:

- `https://golang.org/src/runtime/proc.go` refers to `https://golang.org/s/go11sched`, which is surprisingly rough
- `http://morsmachine.dk/go-scheduler` describes the design document
- `https://golang.org/doc/go1.2`
  - "1.2 is a smaller delta than the step from 1.0 to 1.1, but it still has some significant developments, including a **better scheduler**"
- `https://golang.org/doc/go1.5`
  - "the order in which goroutines are scheduled has been changed"

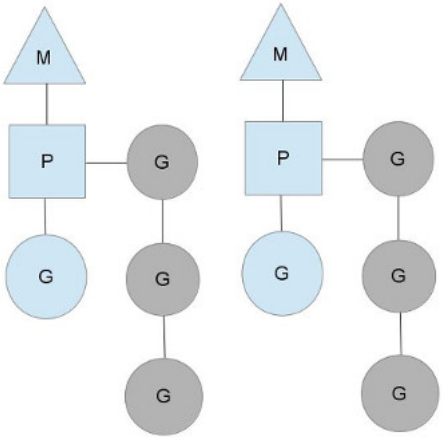
Resources

Three main entities used by the Go scheduler:



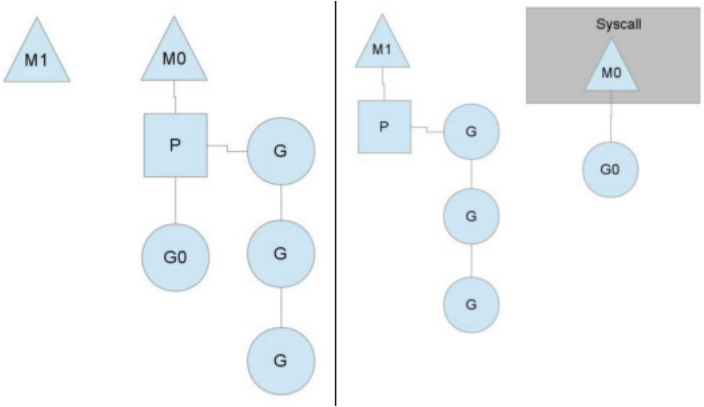
- M: operating system thread
- P: processor/scheduler that runs Go code on a thread
- G: goroutine

Scheduling in action



- M: operating system thread
- P: processor (scheduler) that runs Go code on a thread
- G: goroutine

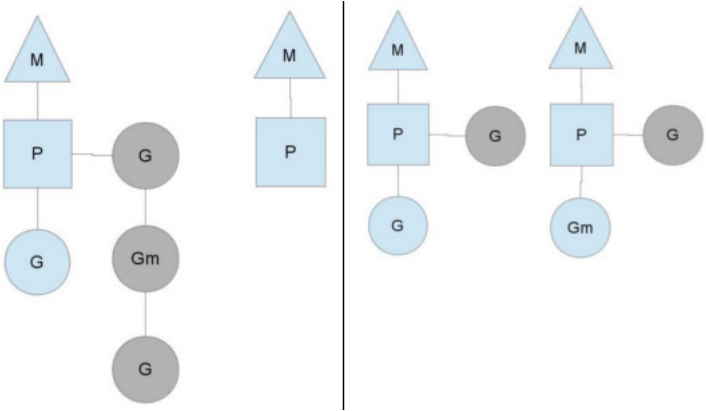
Making a system call



- M: OS thread
- P: processor/scheduler
- G: goroutine

Source: “The Go scheduler” by Daniel Morsing is licensed under CC BY 3.0.

Stealing



- M: OS thread
- P: processor/scheduler
- G: goroutine

GOMAXPROCS

From https://golang.org/pkg/runtime/...

The GOMAXPROCS variable limits the number of operating system threads that can execute user-level Go code simultaneously. There is no limit to the number of threads that can be blocked in system calls on behalf of Go code; those do not count against the GOMAXPROCS limit.

GOMAXPROCS:

- Go $\leq$1.4: default was 1
- Go $\geq$1.5: default is CPU count

Question

```
package main

const LoopCount int = 50

func main() {
        for i := 0; i < LoopCount-1; i++ {
                go func() {
                        for {
                        }
                }()
        }

        for {
        }
}
```

Summary

We've seen…

- processes → lots of overhead
- threads (or lightweight processes) → significant overhead
- goroutines → less overhead

Thanks!

boersn@macewan.ca