Goroutines and concurrency

Goroutines

There is only one operation: `go` (launch/fork/spawn etc).

General form:

```
...
go ... (...)
   |||  ^^^ arguments
   ^^^ function name or anonymous function
```

I.e. the `go` always has to be followed by a function call. Why? No good reason.

Running the goroutine:

- the function application is run in a new parallel process, the "goroutine"
- execution immediately resumes at the statement following the `go` (as opposed to *blocking*)
- the goroutine keeps running until the earliest of
    - function call finishes
    - main program finishes

Example. Run a bunch of factorial computations in parallel: see code.

Channels

Goroutines can communicate two ways

1. *Directly* using shared memory.
2. *Indirectly* using message channels.

A channel has a buffer (queue) of specified size, plus two basic ops.

1. `c<-`
   send to channel `c`; takes one argument, as in `c<-x`.
2. `<-c`
   receive from channel `c`; takes 0 arguments, as in `x := <-c`.

```
var c chan int = make(chan int, 3)
```

Race condition

Assume

1. We have a "system" S running some processes (goroutines, threads etc) in parallel.
2. We have some specification of some system properties, e.g.

- the database has no duplicate keys
- the value of factorial(n) is n!
- If a process is running a program that has `y = x; f(y)` in it, then the call to `f` gets the value retrieved from `x` in the preceding statement

A *race condition* exists if there is some specification which is true in some executions of the system and not in others.

Often the term is used loosely, e.g. when different runs give different values, even if the difference doesn't matter for correctness.