What is a coroutine?

- **Coroutines** are computer program components that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, iterators, infinite lists and pipes. – Wikipedia, the free encyclopedia

- "Subroutines are special cases of ... coroutines." – Donald Knuth. When subroutines are invoked, execution begins at the start and once a subroutine exits, it is finished; an instance of a subroutine only returns once. Coroutines are similar except they can also exit by calling other coroutines, which may later return to the point of calling in the original coroutine; from the coroutine's point of view, it is not exiting at all but simply calling another coroutine.

Coroutine use-cases

- With great power comes great responsibility:
  → Do not do weird things with coroutines. Just use it for concurrency along with the Actor model as Erlang and Go do.
- A context switch between OS threads requires an interrupt or a system call (involving the OS kernel), which can cost more than thousand CPU cycles on x86 CPUs. By contrast, transferring control among coroutines requires only fewer than hundred CPU cycles because it does not involve system calls as it is done cooperatively within a single thread.

Boost Contexts, Coroutines and Fibers

- boost::context implements lightweight cooperative multitasking contexts to switch between coroutines or fibers.
  Each context has its own stack.
- How does it work?
  - make_fcontext
  - jump_fcontext
- Source code by Oliver Kowalke (https://github.com/olk)

Conclusions

- Blocking I/O operations prohibit user-mode scheduling
- No integrated thread pool for scheduling coroutines/fibers
- Go
  → Scalable Goroutine scheduling
- Erlang
  → Scalable reduction count scheduling
  → Blocking operations do not starve the scheduler