libdill: Structured Concurrency for C

What is concurrency?

Concurrency means that multiple functions can run independently of each another. It can mean that they are running in parallel on multiple CPU cores. It can also mean that they are running on a single CPU core and the system is transparently switching between them.

How is concurrency implemented in libdill?

Functions that are meant to run concurrently must be annotated by `coroutine` modifier.

```
coroutine void foo(int arg1, const char *arg2);
```

To launch the function in the same process as the caller use `go` keyword:

```
go(foo(34, "ABC"));
```

To launch it in a separate process use `proc` keyword:

```
proc(foo(34, "ABC"));
```

Following table explains the trade-offs between the two mechanisms:

	go()	**proc()**
lightweight	yes	no
parallel	no	yes
scheduling	cooperative	preemptive
failure isolation	no	yes

Launching a concurrent function -- a `coroutine` in libdill terminology -- using `go` construct is extremely fast, if requires only few machine instructions. In other words, coroutines can be used as flow control mechanism, similar to `if` or `while`. The performance is comparable. `proc`, on the other hand, launches a separate OS process with its own address space and so on. It is much slower.

Same applies to switching between different coroutines. While switching from one coroutine to other inside a single process is super fast, switching between processes is slow. OS scheduler gets involved, TLBs are switched and so on. Context switch between processes is a system hiccup and it often takes many thousands cycles to get back up to speed.

However, there's one huge advantage of using separate processes. They may run in parallel, meaning that they can utilise multiple CPU cores. Launching the coroutine inside of the process, on the other hand, means that it shares CPU with the parent coroutine. If you are using `go` exclusively your program will use only a single CPU core even on a 32-core machine.

From the performance point of view, the best strategy is to have as many processes as there are CPU cores and deal with any remaining concurrency needs inside the process via `go` mechanism.

It's also worth noting that scheduling between different processes is preemptive, in other words that switch from one process to another can happen at any point of time.

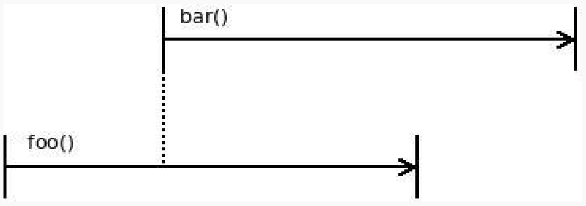
Coroutines within a single process are scheduled cooperatively. What that means is that one coroutine has to explicitly yield control of the CPU to allow a different coroutine to run. In the typical case this is done transparently to the user: When coroutine invokes a function that would block (like `msleep` or`chrecv`) the CPU is automatically yielded. However, if a coroutine does work without calling any blocking functions it may hold the CPU forever. For these cases there's a `yield` function to yield the CPU to other coroutines manually.

Finally, there's a difference in how failures are handled. If a corutine crashes it takes down entire process along with all the other coroutines running inside it. However, if two coroutines are running in two different processes the fact that one of them crashes has no effect on the other one.

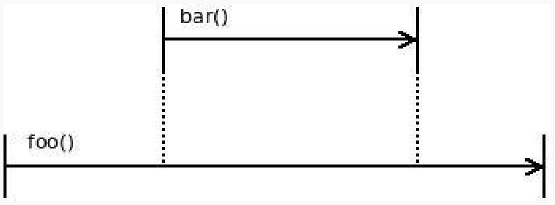
What is structured concurrency?

Structured concurrency means that lifetimes of concurrent functions are cleanly nested one inside another. If coroutine `foo` launches coroutine `bar` then `bar` must finish before `foo` finishes.

This is not structured concurrency:

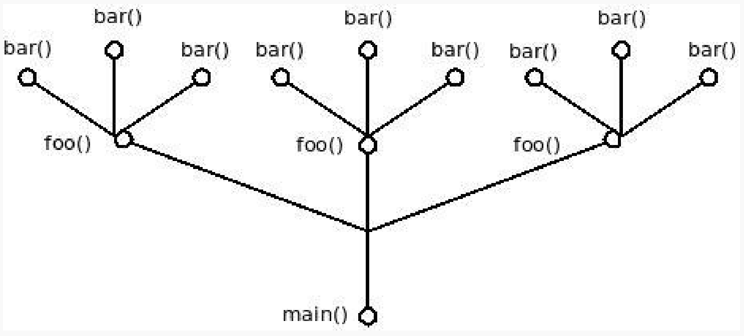


On the other hand, this is structured concurrency:

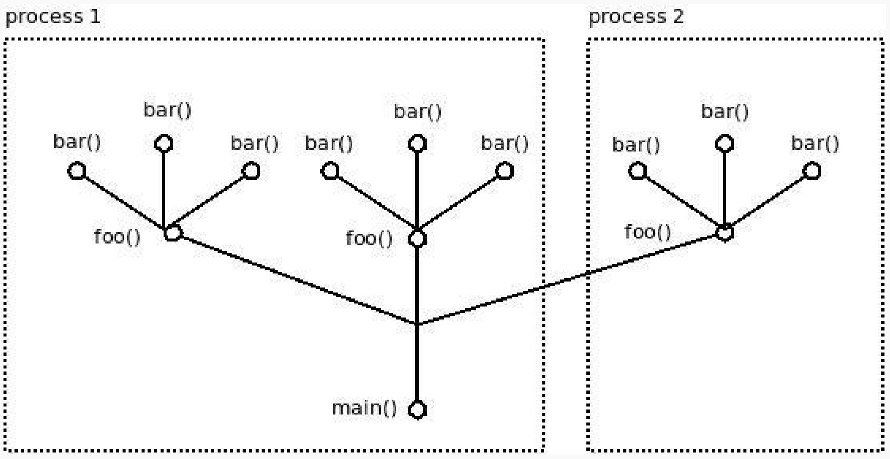


The goal of structured concurrency is to guarantee encapsulation. If main function calls `foo` which in turn launches `bar` in concurrent fashion, main is guaranteed that after `foo` finishes there are no leftover functions still running in the background.

What you end up with is a tree of coroutines rooted in the main function and spreading out toward the smallest worker functions. You also think of it as a generalisation of call stack, a call tree, if you will, in which you can walk from any particular function towards the root until you reach the main function:



It should be noted that the call tree, via `proc` mechanism, can span multiple processes:



How is structured concurrency implemented in

libdill?

As with everything that's idiomatic C you have to do it by hand.

The good news is that it's easy to do.

Both `go` and `proc` return a handle. The handle can be closed thus killing the concurrent function.

```
int h = go(foo());
do_work();
hclose(h);
```

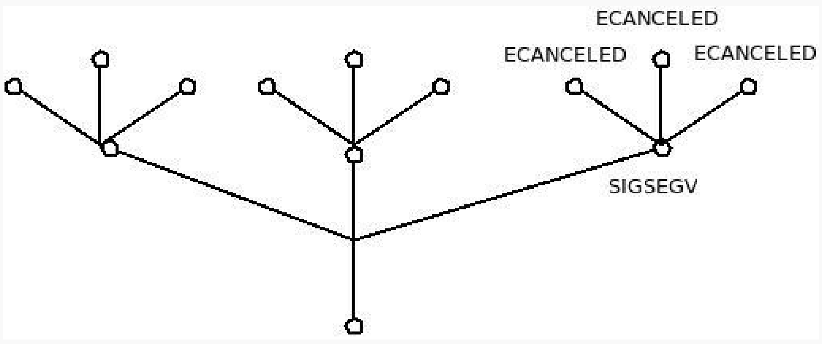
What about function being killed? It may have some resources allocated and we want it to finish cleanly, not leaving any memory or resource leak behind.

The mechanism is simple. In function being killed by `hclose` all the blocking calls start returning `ECANCELED` error. That on one hand forces the function to finish quickly (there's no much you can do without blocking functions anyway) but it also provides a way to clean up:

```
coroutine void foo(void) {
    void *resource = malloc(1000);
    while(1) {
        int rc = msleep(now() + 100);
        if(rc == -1 && errno == ECANCELED) {
            free(resource);
            return;
        }
    }
}
```

Processes launched by `proc` behave similarly. When `hclose` is called they are not killed immediately. Rather, blocking calls in the main coroutine start failing with `ECANCELED` error.

If one of the processes in the process tree crashes or if it is killed, all its child processes behave as if they were closed using `hclose`, i.e. blocking calls in the main coroutine start failing with `ECANCELED` error:

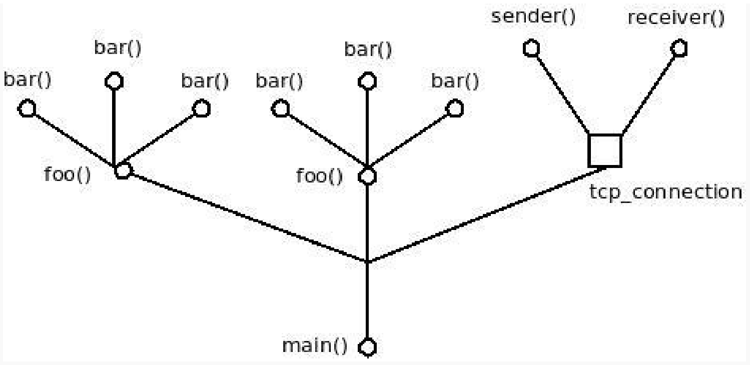


What about asynchronous objects?

Sometimes you don't want to launch a coroutine but rather to create an object that runs coroutines in the background. For example, an object called "tcp_connection" may run two coroutines, one for asynchronously reading data from the network, one for sending data to the network.

Still, it would be nice if the object was a node in the calltree, just like a coroutine is.

In other words, you want a guarantee that once the object is deallocated there are no leftover coroutines running:



And there's no trick there. Just do it in the most straightforward way. Launch the coroutines in function that opens the object and close them in the function the closes the object. Once main function closes the connection object, both sender and receiver corotuines will be automatically stopped.

```
struct tcp_connection {
    int sender;
    int receiver;
}

void tcp_connection_open(struct tcp_connection *self) {
    self->sender = go(tcp_sender(self));
    self->receiver = go(tcp_receiver(self));
}

void tcp_connection_close(struct tcp_connection *self) {
    hclose(self->sender);
    hclose(self->receiver);
}
```