Go-style concurrency in C

Documentation

- Introduction
- A tour of the library
  - Coroutines
  - Channels
  - Choose statement
  - Deadlines
  - Sleeping
  - Waiting for file descriptors
  - Network
    - IP addresses
    - TCP
    - UNIX
    - UDP
  - Coroutine-local storage
  - Dealing with memory
  - Multiprocessing
  - Debugging
- Reference
  - chclose
  - chdone
  - chdup
  - chmake
  - choose
  - chr
  - chs
  - cls
  - coroutine
  - fdclean
  - fdwait
  - go
  - goprepare
  - goredump
  - gotrace
  - ipaddrstr
  - iplocal
  - ipremote
  - mfclose
  - mfeof
  - mferr
  - mfflush
  - mfin
  - mfopen
  - mfork
  - mfout
  - mfread
  - mfseek
  - mftell
  - mfwrite
  - msleep
  - now
  - setcls
  - tcpaccept
  - tcpaddr
  - tcpclose
  - tcpconnect
  - tcpflush

tcplisten
tcpport
tcprecv
tcprecvuntil
tcpsend
udpclose
udplisten
udpport
udprecv
udpsend
unixaccept
unixclose
unixconnect
unixflush
unixlisten
unixpair
unixrecv
unixrecvuntil
unixsend
yield

Introduction

Libmill is a lightweight coroutine library bringing Go-style concurrency to C language. It also contains simple networking and file access library that allows users to quickly bootstrap application development.

Libmill runs in following environments:

- Microarchitecture: x86, x86_64, ARM
- Compiler: gcc, clang
- Operating system: Linux, OSX, FreeBSD, OpenBSD, NetBSD, DragonFlyBSD

Whether it works in different environments is not known - please, do report any successes or failures to the project mailing list.

Download the package from the download page. Then build and install as follows:

```
$ tar -xzf libmill-1.18.tar.gz
$ cd libmill-1.18
$ ./configure
$ make
$ make check
$ sudo make install
```

After the installation there's `libmill.h` header file available as well as dynamic library `libmill.so` and static library `libmill.a`.

A program using the library can look, for example, like this:

```
#include <stdio.h>
#include <libmill.h>

coroutine void worker(int count, const char *text) {
    int i;
    for(i = 0; i != count; ++i) {
        printf("%s\n", text);
        msleep(now() + 10);
    }
}

int main() {
    go(worker(4, "a"));
    go(worker(2, "b"));
    go(worker(3, "c"));
    msleep(now() + 100);
    return 0;
}
```

To build and run the example:

```
$ gcc -o example example.c -lmill
$ ./example
```

Alternatively, using static library:

```
$ gcc -o example example.c libmill.a
$ ./example
```

Libmill defines some fairly generic symbols like `in` or `go`. If it clashes with other libraries you can define `MILL_USE_PREFIX` macro before including the header file. That will add `mill_` (or `MILL_`) prefix to all the defined symbols. For example:

```
#define MILL_USE_PREFIX
#include <libmill.h>

mill_coroutine void worker(void) {
}

int main() {
    mill_go(worker());
    return 0;
}
```

A tour of the library

Coroutines

Coroutine is defined using `coroutine` keyword and launched using `go` construct:

```
coroutine void foo(int i, int j) {
    ...
}

int main() {
    go(foo(1, 2));
    return 0;
}
```

Although coroutine can return a value keep in mind that it is executed asynchronously and thus there is now way to retrieve the result.

Note that your program may work perfectly well even if you ommit the `coroutine` keyword. However, such program may break randomly when compiled by a different compiler or with a different optimisation level (a known problem is with clang and -O2 optimisation level).

Also bear in mind that coroutines are scheduled in a cooperative fashion. If one coroutine blocks, for example by calling `sleep()`, it blocks the entire process. Therefore, coroutine-friendly versions of blocking functions (such as `msleep()` or `fdwait()`, see below) should be used.

In case of need, you can yield CPU to other coroutines explicitly:

```
yield();
```

Channels

Channels are typed uni-directional pipes used for communication between coroutines. To create a channel use `chmake()` function like this:

```
chan ch = chmake(int, 0);
```

This channel can be used to pass integer values. However, one can declare channel of any type.

The second parameter is the size of the channel's buffer. If it is set to zero, it means that the channel is unbuffered. Sender will block until there's a receiver available and vice versa.

If the channel has buffer size set to 100 the sender can send 100 messages without anyone receiving them. When trying to send 101st message it will block.

To send a message to the channel use `chs()` function:

```
chs(ch, int, 42);
```

To receive a message from the channel use `chr()` function:

```
int i = chr(ch, int);
```

When sender is done with sending it can announce the fact using `chdone()` function:

```
chdone(ch, int, -1);
```

The value (third parameter) passed to the function is used to signal to the receivers that sender is done. Once all the actual messages are read from the channel any subsequent calls to `chr()` won't block and will return the specified message instead. The code on the receiver side typically looks like this:

```
while(1) {
    int i = chr(ch, int);
    if(i == -1)
        break;
    ...
}
```

To deallocate the channel use `chclose()` function:

```
chclose(ch);
```

Note that `chclose()` deallocates any elements remaining the channel, however, if the element itself has a pointer to a different object, that object won't get deallocated. In this respect `chclose()` is similar to the standard C `free()` function.

When sharing a channel between two or more coroutines you can duplicate the channel handle. Channel gets deallocated only after all its handles are closed. For example:

```
coroutine void sender(chan ch) {
    chs(ch, int, 42);
    chclose(ch);
}

int main() {
    chan ch = chmake(int, 0);
    go(sender(chdup(ch)));
    int i = chr(ch, int);
    assert(i == 42);
    chclose(ch);
    return 0;
}
```

Choose statement

Choose statement is equivalent to Go's `select` statement. It can be used to wait for multiple channels.

Inside `choose` statement there is a list of clauses. Each clause is either an `in` clause (waiting for an incoming message) or an `out` clause (waiting till a message can be sent to the channel). Last clause can optionally be a

`deadline` or an `otherwise` clause. The former gets executed after deadline expires, the latter if none of the other clauses apply. Don't forget to add the `end` keyword before the closing parenthesis:

```
choose {
in(ch1, int, val):
    printf("Value %d received from ch1.\n", val);
out(ch2, int, 42):
    printf("Value 42 sent to ch2.\n");
otherwise:
    printf("Neither ch1 nor ch2 can be used at the moment.\n");
end
}
```

Clause `in` causes `choose` to wait for a message coming from the channel. The clause declares variable `var` of type `type` and sets it to the value of the incoming message.

```
in(ch, type, var):
```

Clause `out` causes `choose` to wait while value `val` of type `type` can be sent to channel `ch`. Same way as with in, if `type` doesn't match the actual type of channel `ch` runtime exception is generated:

```
out(ch, type, val):
```

Clause `deadline`, if present, is executed after deadline expires and none of the other clauses fire before that:

```
deadline(now() + 1000):
```

Clause `otherwise` is optional. It is executed only if no other clauses can be matched immediately. Therefore, `choose` statement with `otherwise` clause will never block.

`choose` statement waits for the first clause that can be executed, executes it and then jumps to the code following the clause. If there are multiple clauses that can be executed one of them is chosen at random.

There's no fall-through among the clauses.

Deadlines

Unlike standard POSIX functions, libmill uses deadlines rather than timeouts.

Deadline is a point in time when a function should finish. Function `now()` returns the current point in time. You can add milliseconds to it to get a point in time in the future. Following example defines a deadline that expires one second from now:

```
int64_t deadline = now() + 1000;
```

The advantage of deadlines over timeouts is that a single deadline can be reused in many subsequent function calls without having to worry about readjusting its value to account for time already elapsed since the deadline was created.

Value of -1 is used to specify that there is no deadline or, if you will, that the deadline is infinite.

Sleeping

It was already mentioned that a single call to `sleep()` can block the entire process. To avoid the problem use `msleep()` instead. Note that the argument is a deadline, not a timeout:

```
msleep(now() + 1000);
```

Waiting for file descriptors

To wait for an event from a file descriptor use fdwait function:

```
int events = fdwait(fd, FDW_IN | FDW_OUT, -1);
if(events & FDW_IN) {
    ...
}
if(events & FDW_OUT) {
    ...
}
if(events & FDW_ERR) {
    ...
}
```

FDW_IN waits until there's at least one byte available to be read from the file descriptor.

FDW_OUT waits until at least one byte can be written to the file descriptor.

FDW_ERR cannot be used as an argument to fdwait(), but it can be returned from the function irrespective of whether you want it or not.

If an error happens while there are still bytes to be received from the socket combination of FDW_ERR and FDW_IN may be returned.

The third parameter to fdwait() is a deadline. If deadline is hit the function returns zero.

When handling the file descriptors directly (as opposed to using tcpsock, unixsock or udpsock) you have to call fdclean() before closing any file descriptor you've previously used with fdwait(). It will drop any cached state fdwait() may have associated with the descriptor. If you don't do so undefined, hard to debug behaviour is likely to ensue.

```
fdclean(fd);
close(fd);
```

As for the file descriptors you don't own, as, for example, file descriptors temporarily handed to you by a third party library you should use fdclean() before returning the ownership to the original owner.

Network

Network functionality, such as TCP or UDP, can be accessed via standard POSIX functions provided that you do it in a non-blocking way. To save you the trouble though, libmill provides a simple convenience wrapper on top of it.

IP addresses

To use TCP or UDP you need IP addresses first. The wrapper type for IP addresses (both IPv4 and IPv6) is called ipaddr. It also includes port number. Unlike address types in POSIX it can be used as a basic type. It can be assigned, returned from a function and so on.

There are two functions to construct IP addresses: iplocal() that retrieves the IP address of a local network interface and ipremote() that gets IP address of a remote machine. In addition to that, certain other library functions, (udprecv(), tcpaddr()) return IP addresses.

iplocal() converts a textual form of IP address, or a name of a local network interface and a port number into ipaddr:

```
ipaddr addr1 = iplocal("127.0.0.1", 3333, 0);
ipaddr addr2 = iplocal("::1", 4444, 0);
ipaddr addr3 = iplocal("eth0", 5555, 0);
```

The last (mode) argument specifies which kind of addresses you are interested in. Possible values are:

- IPADDR_IPV4: get IPv4 address or error if it's not available
- IPADDR_IPV6: get IPv6 address or error if it's not available
- IPADDR_PREF_IPV4: get IPv4 address if possible, IPv6 address otherwise; error if none is available
- IPADDR_PREF_IPV6: get IPv6 address if possible, IPv4 address otherwise; error if none is available

Setting the argument to zero means default behaviour, which, at the present, is IPADDR_PREF_IPV4. However, in the future when IPv6 becomes more common it may be switched to IPADDR_PREF_IPV6.

If address parameter is set to NULL, `INADDR_ANY` or `in6addr_any` is returned. This value is useful when binding to all local network interfaces.

The second function to produce IP addresses is `ipremote()`. It resolves remote host name to an IP address.

Given that remote host name resolution may involve a DNS query which in turn can take non-trivial amount of time, `ipremote()` has also deadline parameter.

```
ipaddr addr = ipremote("192.168.0.111", 5555, 0, -1);
```

Both `iplocal()` and `ipremote()` report errors in the same way. They set `errno` to appropriate error code. Additionally, returned address has family type AF_UNSPEC which will cause any function you pass it to fail.

You can also convert `ipaddr` into a human readable string. To do so you have to supply a buffer at least `IPADDR_MAXSTRLEN` long:

```
ipaddr addr = ...;
char buf[IPADDR_MAXSTRLEN];
printf("Connecting to %s!\n", ipaddrstr(addr, buf));
```

TCP

Connecting to a remote endpoint

To connect to a remote server, use `tcpconnect()` function:

```
ipaddr addr = ipremote("192.168.0.111", 5555, 0, -1);
tcpsock s = tcpconnect(addr, -1);
```

Listening for incoming connections

To start listening for incoming TCP connections use `tcplisten()` function. To accept a new connection use `tcpaccept()` function:

```
ipaddr addr = iplocal("eth0", 5555, 0);
tcpsock ls = tcplisten(addr, 10);
while(1) {
    tcpsock s = tcpaccept(ls, -1);
    ...
}
```

First argument is an IP address of the local network interface to bind to. Use `iplocal(NULL, port, 0)` to bind to all local interfaces.

If port number is zero an ephemeral port number will be chosen by the operating system. In such case you can retrieve the port number using `tcppport()` function:

```
ipaddr addr = iplocal(NULL, 0, 0);
tcpsock ls = tcplisten(addr, 10);
int port = tcpport(ls);
```

`tcpaccept()` function takes the listening socket as argument and returns the newly accepted socket:

```
tcpsock s = tcpaccept(ls, -1);
```

After accepting new connection you can retrieve IP address of the peer using `tcpaddr()` function:

```
tcpsock s = tcpaccept(ls, -1);
ipaddr peer = tcpaddr(s);
```

Sending data

To send data to the connection use tcpsend() function:

```
size_t nbytes = tcpsend(s, "ABC", 3, -1);
```

Number of bytes actually sent is returned. Given that libmill functions generally work in blocking-like fashion the result will be equal to number of bytes requested, unless an error has occured.

To optimise the throughput tcpsend() may store the data into output buffer and not flush it into the network immediately. To force actual sending of the data use tcpflush() function.

```
tcpflush(s, -1);
```

Keep in mind that not flushing the data can mean that it will never appear at the peer.

Receiving data

To read data from a connection use tcprecv() function:

```
char buf[256];
size_t nbytes = tcprecv(s, buf, sizeof(buf), -1);
```

Similarly as with tcpsend(), tcprecv() returns number of bytes received and the value should match the number of bytes requested unless there is an error or deadline is reached.

To help with dealing with text-based protocols libmill provides tcprecvuntil() function. It acts the same as tcpread() except that the reading stops when one of specified characters is encountered:

```
char buf[256];
size_t nbytes = tcprecvuntil(conn, buf, sizeof(buf), "\n" , 1, -1);
```

The arguments are the socket to receive from, the buffer to receive to, size of the buffer, the delimiter characters, how many of them are there and a deadline.

Number of bytes actually received, including the delimiter character, if found, is returned. If the entire buffer was filled without encountering the delimiter character, errno is set to ENOBUFS.

Keep in mind that mutliple delimiter characters means that receiving stops if any of them is received rather than when the entire string of them is received:

```
/* Read until comma or right brace is encountered. */
size_t nbytes = tcprecvuntil(conn, buf, sizeof(buf), ",}" , 2, -1);
```

Cleaning up

To close sockets (both listening and connected ones) use tcpclose() function:

```
tcpclose(s);
```

UNIX

Unix sockets are used for communicating between processes on the same machine.

The API is identical to the TCP API with all `tcp` prefixes replaced by `unix` prefixes. (For example `unixsock` or `unixrecv()`.)

There are few minor adjustments to the API though.

Addresses passed to `unixlisten()` and `unixconnect()` are names of a local files rather than IP addresses. Also, there are no ports:

```
unixsock s = unixconnect("/tmp/test.unix");
```

Function `unixpair()` creates a pair of connected sockets. It's a wrapper on top of standard `socketpair()` function:

```
unixsock s1, s2;
unixpair(&s1, &s2);
```

UDP

To open an UDP socket use `udplisten()` function. IP address of the local network interface should be supplied:

```
ipaddr addr = iplocal("192.168.0.111", 5555, 0);
udpsock s = udplisten(addr);
```

If port is set to zero operating system selects an unused port. The number of the port can be retrieved using `udpport()` function:

```
udpsock s = udplisten(addr);
int port = udpport(s);
```

Once the socket is open it can be immediately used for both sending and receiving.

Given that UDP transport is unconnected by its very nature the address of remote endpoint has to be specified in every `udpsend()` call. (Similarly, the IP address of the remote endpoint is returned by each `udprecv()` call.)

```
ipaddr outaddr = ipremote("192.168.0.111", 5555, 0, -1);
udpsend(s, outaddr, "Hello, world!", 13);
...
char buf[256];
ipaddr inaddr;
size_t sz = udprecv(s, &inaddr, buf, sizeof(buf), -1);
```

Note that `udpsend()` has no deadline. UDP transport is unreliable and the packet is either delivered or lost, but never blocked.

When done close the UDP socket:

```
udpclose(s);
```

Coroutine-local storage

Coroutine-local storage is an advanced mechanism meant to be used in special cases. Avoid using it if possible. In general, use it where you would use thread-local storage in a threaded program. TLS itself cannot be used with libmill as multiple coroutines can share the same OS thread.

Coroutine local data are a single pointer which can be set like this:

```
void *data = ...;
setcls(data);
```

Later on the pointer can be accessed anywhere within the coroutine or any function invoked from it using `cls` function:

```
void *data = cls();
```

It can be safely assumed that before coroutine-local data are set, `cls` function will return `NULL`.

Dealing with memory

Most libmill functions report out of memory condition in the common way. `chmake()` returns `NULL`, `tcplisten()` returns `NULL` and sets `errno` to `ENOMEM` and so on.

There's one exception though. Namely, if there's not enough memory to allocate call stack for a new coroutine runtime panic happens and the process is terminated.

If you wish to prevent that from happening you can allocate the coroutine stacks in advance, typically when the process is starting, using `goprepare()` function.

Specify the number of coroutine stacks to preallocate, their size in bytes and, additionally, the size of the biggest element you are going to pass through libmill channels:

```
goprepare(100, 100000, 128);
```

From that point on your program shouldn't fail when launching a coroutine. However, if you launch more parallelly running coroutines than allocated or pass a bigger element than specified through a channel, memory allocations will start happening again and all bets are off.

NOTE: On some systems (e.g. Linux) memory overcommit is switched on by default. On such systems your application may be selected to be killed in low memory conditions without you having any say in it. If you can't guarantee that memory overcommit will be switched off when running the program don't even bother with the measures described above.

Multiprocessing

Libmill is intended for writing single-threaded applications. If you want to take advantage of multi-core machines follow the UNIX way and use multiprocessing.

An example can be found here.

Debugging

Libmill provides a couple of functions to help with the debugging. They can be invoked either directly from the program or manually from the debugger.

When inspecting debug output following rules apply:

1. Both coroutines and channels are represented by decimal IDs.
2. The IDs are never re-used, even if the original object was already deallocated.
3. Coroutine IDs are printed in curly braces, e.g. `{24}`.
4. Channel IDs are printed in pointy braces, e.g. `<24>`.

goredump

When debugging a program you may want to inspect the coroutines being run and the channels being used. Use `goredump()` function to do exactly that.

Here's an example of how to use it directly from a `gdb` session:

```
Breakpoint 1, main () at foo.c:57
57          if(quux < 0) {
(gdb) p goredump()

COROUTINE  state                  current           created
```

```
------------------------------------------------------------------------
{0}        RUNNING               ---               <main>
{7}        chs(<7>)              foo.c:63          foo.c:157
{8}        chr(<8>)              foo.c:63          foo.c:158
{14}       choose(<14>,<14>)     foo.c:70          foo.c:258
{15}       ready                 foo.c:35          foo.c:276

CHANNEL  msgs/max    senders/receivers     refs  done  created
------------------------------------------------------------------------
<7>      0/0         s:{7}                 2     no    foo.c:155
<8>      0/0         r:{8}                 2     no    foo.c:156
<14>     0/0         s:{14},{14}           2     no    foo.c:257
<15>     22/100                            1     yes   foo.c:274

$2 = void
(gdb)
```

The first section is coroutine dump. It contains the list of all coroutines, each accompanied by its ID, state, the line of code that's it is executing at the moment and to the line that originally created the coroutine.

The dump doesn't contain coroutine name. To map the coroutine record to the actual coroutine in the code, check the line referenced by 'created' field.

The 'state' field gives you more information about the operation being performed. When a coroutine is blocked in `chr()` call, for example, you'll also see the ID of the channel it is trying to receive from:

```
COROUTINE  state                 current           created
------------------------------------------------------------------------
{8}        chr(<8>)              foo.c:63          foo.c:158
```

When `choose` statement is being executed a list of channel IDs is provided, corresponding to the list of clauses in the `choose` statement:

```
COROUTINE  state                 current           created
------------------------------------------------------------------------
{14}       choose(<5>,<3>)       foo.c:22          foo.c:107
```

If there are any channels created by the user coroutine dump will be followed by a channel dump. Each record in the dump corresponds to one channel instance and contains its ID, number of messages buffered in the channel, channel capacity, its reference count, whether `chdone()` was already called and a the line of code that created the channel.

Additionally, if there are any coroutines waiting for messages from the channel -- whether via `chr` or `choose` -- they are reported like this:

```
CHANNEL  msgs/max    senders/receivers     refs  done  created
------------------------------------------------------------------------
<1>      0/0         r:{4},{5}             2     no    foo.c:391
```

The waiting coroutines are served in the displayed order, i.e. when a new message is sent to the channel it will be dispatched to coroutine 4. Next one will go to coroutine 5 and so on.

Similarly, if there are coroutines trying to send message to the channel but unable to do so becuase the channel's buffer if full they are reported in the following way:

```
CHANNEL  msgs/max    senders/receivers     refs  done  created
------------------------------------------------------------------------
<1>      100/100     r:{2},{13}            2     no    foo.c:392
```

If there are no coroutines waiting to send or receive messages from the channel the field is kept empty.

gotrace

It is possible to switch on tracing via `gotrace()` function. You can do so either programmatically or from the debugger:

```
gotrace(1);
```

The function takes a single argument, the desired tracing level. At the moment there are only two supported tracing levels:

- 0: tracing is off; this is the default value.
- 1: invocations of all libmill functions are traced.

Trace records are written to `stderr`. That way it is possible to redirect the trace records to a file, while keeping the original output of the program in the console:

```
$ ./test 2> trace.log
```

A part of the trace may look, for example, like this:

```
==> 22:24:56.404183 {0}      <1>=chmake(0) at tests/choose.c:113
==> 22:24:56.404276 {0}      {1}=go() at tests/choose.c:114
==> 22:24:56.404292 {1}      chdup(<1>) at tests/choose.c:114
==> 22:24:56.404304 {1}      chr(<1>) at tests/choose.c:30
==> 22:24:56.404316 {0}      choose() at tests/choose.c:115
==> 22:24:56.404354 {0}      chclose(<1>) at tests/choose.c:120
```

Each trace record contains timestamp, ID of the coroutine that have generated the trace, the function being executed along with its arguments and the location of its invocation in the source code.

The format is designed to be machine-friendly and, more specifically, grep-friendly.

If you have a file that mixes trace records with other output, grep will help you filter out the trace records:

```
$ grep "==>" trace.log
```

Similarly, you can filter out records related to a particular coroutine or a specific channel:

```
$ grep "{9}" trace.log
$ grep "<33>" trace.log
```

Reference

void chclose(chan ch);

Closes a channel handle. After last copy of the handle (see `chdup()`) is closed, channel is deallocated.

Keep in mind that deallocation of a channel works very much like deallocation of local variables when function returns. The memory holding the variable itself won't leak, however, if the variable contains a pointer to a different object that one doesn't get deallocated.

The function cannot fail.

void chdone(chan ch, type, type value);

This function is used to announce that the sender will send no more messages to the channel.

First argument is the handle to the channel. Second one is type of the termination message and last one is the termination message itself. The type of the message must match the type of the channel otherwise the program will panic and abort.

When all messages are retrieved from the channel, all the receivers will receive the termination message over and over again.

The function cannot fail.

chan chdup(chan ch);

Duplicates channel handle. Channel gets deallocated only after all its handles are closed using `chclose()` function.

The function returns duplicated channel handle and cannot fail.

chan chmake(type, size);

Creates a channel. First parameter is the type of the items to be sent through the channel. Second parameter is number of items that will fit into the channel. Setting this argument to zero creates an unbuffered channel.

The function returns channel handle or NULL in case there's not enough memory to allocate the channel.

choose {}

Waits for activity on multiple channels.

The statement consists of multiple clauses. Last clause must always be `end`:

```
choose {
in(ch1, int, i):
    foo(i);
in(ch2, int, i):
    bar(i);
out(ch3, int, 42):
    baz();
otherwise:
    quux();
end
}
```

`in` clause waits for a message coming from a channel. It accepts three arguments. The channel to receive the message from, the type of the message and the variable to assign the message to.

The variable is declared by the clause, you don't have to declare it beforehand yourself.

If message type doesn't match the channel type program will panic and abort.

`out` clause waits until a message can be sent to a channel. It may not be immediately possible as the channel may be full. The clause accepts three arguments. The channel to send the message to, the type of the message and the message itself.

If message type doesn't match the channel type program will panic and abort.

`deadline` clause will be executed if deadline expires while waiting for other clauses. For example:

```
choose {
in(ch1, int, i):
    foo(i);
deadline(now() + 1000):
    printf("Deadline exceeded!\n");
end
}
```

`otherwise` clause is run if no other clause can be executed immediately. A `choose` statement with `otherwise` clause therefore never blocks.

`deadline` and `otherwise` clause are mutually exclusive and neither of them can occur multiple times.

If multiple clauses can be executed immediately, one of them is chosen at random.

There's no fall-through among the clauses. In other words, there's no need to put `break` at the end of each clause as is the case with standard C `switch` statement.

If `break` statement is used it will break out of the `choose` construct nonetheless. Behaviour of `continue` statement within `choose` is undefined.

The construct returns no errors.

type chr(chan ch, type);

Retrieves a message from the channel. First parameter is the channel handle, second one is the type of the message. The type specified must match the type of the channel otherwise the program will panic and abort.

If there's no message in the channel the function waits until one becomes available.

Retrieved message is returned from the function. It cannot fail.

void chs(chan ch, type, type value);

Send a message to the channle. First parameter is the channel handle, second one is the type of the message and third one is the message itself. If the type of the message doesn't match the type of the channel the program will panic and abort.

If there's no space in the channel for the message the function waits until it becomes available.

Sending to a channel that was previously terminated using `chdone()` function results in runtime panic.

The function cannot fail.

void *cls(void);

Returns coroutine-local storage, a single pointer that is accessible from anywhere within the same coroutine.

You can set the coroutine-local storage using `setcls()` function. If it was not previously set `cls()` will return `NULL`.

The function cannot fail.

coroutine

A specifier that must be added to all functions that are executed using `go()`, for example:

```
coroutine void foo(int i);
```

Be aware that programs typically work even without `coroutine` specifier, however, they may fail in random fashion, depending on a particular combination of compiler, optimisation level and code in question.

void fdclean(int fd);

This function drops any cached state associated with the file descriptor. It has to be called before the file descriptor is closed. If it is not, undefined behaviour may ensue.

It should also be used when you are temporarily provided with a file descriptor by a third party library, just before returning the descriptor back to the original owner.

`tcpsock`, `unixsock` and `udpsock` take care of calling this function, so unless you are creating custom file descriptors by hand, you don't have to care.

The function cannot fail.

int fdwait(int fd, int events, int64_t deadline);

Waits for an event from a file descriptor.

The arguments are the file descriptor, the events to wait for and the deadline.

```
int events = fdwait(fd, FDW_IN | FDW_OUT, -1);
```

- FDW_IN: Wait until data can be received from the file descriptor.
- FDW_OUT: Wait until data can de sent to the file descriptor.

`fdwait` returns combination of following flags:

- FDW_IN: Data can be received from the file descriptor.
- FDW_OUT: Data can be sent to the file descriptor.
- FDW_ERR: Error occured.

Note that `FDW_ERR` cannot be used as an argument to the function, but it can be returned irrespective of whether the caller asked for it or not.

If an error happens while there are still data to be received from the file descriptor combination of `FDW_ERR` and `FDW_IN` may be returned.

If deadline is reached while waiting for the file descriptor, zero is returned.

void go(expression);

Launches a coroutine that executes the function passed in the `expression` argument, for example:

```
go(foo(42));
```

Coroutine is executed in concurrent manner and its lifetime may exceed the lifetime of the caller. Therefore, the return value of the function is discarded and cannot be retrieved by the caller.

Any function to be invoked unsing `go()` must be declared with `coroutine` specifier.

Be aware that programs typically work even without `coroutine` specifier, however, they may fail in random fashion, depending on a particular combination of compiler, optimisation level and code in question.

Also, all arguments to a coroutine must not contain function calls. If they do the program may fail non-deterministically. If you need to pass a result of a computation to a coroutine do the computation first, then pass the result as an argument. Instead of:

```
go(bar(foo(a)));
```

Do this:

```
int a = foo();
go(bar(a));
```

If there is not enough memory to allocate new stack for the coroutine, `go()` aborts the process. To avoid that happening you can pre-allocate the stacks on process startup using `goprepare()`.

void goprepare(int count, size_t stack_size, size_t val_size);

Preallocate coroutine stacks. This function is useful to guarantee that coroutine invocation won't fail when there's insufficient memory to allocate the stack.

NOTE: On some systems (e.g. Linux) memory overcommit is switched on by default. On such systems your application may be selected to be killed in low memory conditions without program having any say in it. If you can't guarantee that memory overcommit will be switched off then using `goprepare()` has basically zero effect.

The arguments are number of coroutine stacks to preallocate, their size in bytes and, additionally, the size of the biggest element you are going to pass through libmill channels:

```
goprepare(100, 100000, 128);
```

Program can use up to the specified number of coroutines in parallel without the need for memory allocation. However, if the limit is reached -- or if message larger than `val_size` is sent through a channel -- memory allocation may occur, fail and cause panic and termination of the process.

Additionally, the function preallocates any process-wide resources needed, such as, for example, global pollsets.

The function sets `errno` to zero in case of success or to one of the following errors otherwise:

`EAGAIN`: There are coroutines running. The stacks can't be reallocated.
`EMFILE`: The maximum number of file descriptors in the process are already open.
`ENFILE`: The maximum number of file descriptors in the system are already open.
`ENOMEM`: Not enough memory to allocate the stacks.

void goredump(void);

Dumps current state of all coroutines and all channels to `stderr`.

The function can be launched either from the program of from the debugger.

For more information look here.

void gotrace(int level);

Sets the tracing level.

The function can be launched either from the program of from the debugger.

For more information look here.

const char *ipaddrstr(ipaddr addr, char *buffer);

Formats `ipaddr` as human-readable string.

First argument is the IP address to format, second is the buffer to store the result it. The buffer must be at least `IPADDR_MAXSTRLEN` bytes long.

The function returns pointer to the formatted string.

ipaddr iplocal(const char *name, int port, int mode);

Converts an IP address in human-readable format, or a name of a local network interface, into an `ipaddr` object:

```
ipaddr a1 = iplocal("127.0.0.1", 3333, 0);
ipaddr a2 = iplocal("::1", 4444, 0);
ipaddr a3 = iplocal("eth0", 5555, 0);
```

First argument is the string to convert. Second argument is the port number. Third argument specifies which kind of addresses should be returned. Possible values are:

- IPADDR_IPV4: get IPv4 address or error if it's not available
- IPADDR_IPV6: get IPv6 address or error if it's not available
- IPADDR_PREF_IPV4: get IPv4 address if possible, IPv6 address otherwise; error if none is available
- IPADDR_PREF_IPV6: get IPv6 address if possible, IPv4 address otherwise; error if none is available

Setting the argument to zero invokes default behaviour, which, at the present, is IPADDR_PREF_IPV4. However, in the future when IPv6 becomes more common it may be switched to IPADDR_PREF_IPV6.

If address parameter is set to NULL, INADDR_ANY or in6addr_any is returned. This value is useful when binding to all local network interfaces.

In case of error `errno` is set to appropriate error code, as indicated below. Returned address will then have family type of AF_UNSPEC which in turn will cause any function you pass it to fail with `EAFNOSUPPORT` error.

As the functionality is not covered by POSIX and rather implemented by OS-specific means there's no definitive list of possible error codes the function can return. You have to make the error handling as generic as possible.

ipaddr ipremote(const char *name, int port, int mode, int64_t deadline);

Converts an IP address in human-readable format, or a name of a remote host into an `ipaddr` object:

```
ipaddr a1 = ipremote("192.168.0.111", 5555, 0, -1);
ipaddr a2 = ipremote("www.expamle.org", 80, 0, -1);
```

First argument is the string to convert. Second argument is the port number. Third argument specifies which kind of addresses should be returned. Possible values are:

- IPADDR_IPV4: get IPv4 address or error if it's not available
- IPADDR_IPV6: get IPv6 address or error if it's not available
- IPADDR_PREF_IPV4: get IPv4 address if possible, IPv6 address otherwise; error if none is available
- IPADDR_PREF_IPV6: get IPv6 address if possible, IPv4 address otherwise; error if none is available

Setting the argument to zero invokes default behaviour, which, at the present, is IPADDR_PREF_IPV4. However, in the future when IPv6 becomes more common it may be switched to IPADDR_PREF_IPV6.

Finally, the fourth argument is the deadline. It allows to deal with situations where resolving a remote host name requires a DNS query and the query is taking substantial amount of time to complete.

In case of error `errno` is set to appropriate error code, as indicated below. Returned address will then have family type of AF_UNSPEC which in turn will cause any function you pass it to fail with `EAFNOSUPPORT` error.

As the functionality is not covered by POSIX and rather implemented by OS-specific means there's no definitive list of possible error codes the function can return. You have to make the error handling as generic as possible.

void mfclose(mfile f);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

int mfeof(mfile f);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

mferr

`mfile` wrapper on top of stderr.

void mfflush(mfile f, int64_t deadline);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

mfin

`mfile` wrapper on top of stdin.

mfile mfopen(const char *pathname, int flags, mode_t mode);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

pid_t mfork(void);

This function is a wrapper on top od the standard POSIX `fork()` function. In addition to forking the process it properly detaches libmill's internal resources (such as any kernel-side kqueue/epoll pollsets) in child process from those in the parent process.

Using `fork()` instead of `mfork()` results in undefined behaviour.

The function returns the process ID of the newly created process to the parent process, zero to the child process or -1 in case of error. In the latter case it also sets `errno` to the appropriate error code.

mfout

`mfile` wrapper on top of stdout.

size_t mfread(mfile f, void *buf, size_t len, int64_t deadline);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

off_t mfseek(mfile f, off_t offset);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

off_t mftell(mfile f);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

mfwrite(mfile f, const void *buf, size_t len, int64_t deadline);

WARNING: Please note that file operations, if blocked, will block all the running coroutines.

void msleep(int64_t deadline);

Waits until the `deadline`. Following statement, for example, waits for one second:

```
msleep(now() + 1000);
```

The function returns no errors.

int64_t now(void);

Returns current time, in milliseconds.

The function is meant to be used for creating deadlines. For example, a point of time one second on from now can be expressed as `now() + 1000`.

Value of -1 cannot be returned from the function and is therefore used to mean 'infinite'.

The function cannot fail.

void setcls(void *val);

Set coroutine-local storage, a single pointer that will be accessible from anywhere within the same coroutine using `cls()` function.

The function cannot fail.

tcpsock tcpaccept(tcpsock s, int64_t deadline);

Accepts new incoming TCP connection from `listening` socket `s`. Second argument is the `deadline` for the operation.

```
ipaddr addr = iplocal("127.0.0.1", 5555, 0);
tcpsock listener = tcplisten(addr, 10);
tcpsock connection = tcpaccept(listener, -1);
```

If the function succeeds it returns handle of the newly created connection and sets `errno` to zero.

If the function fails it returns `NULL` and sets `errno` to one of the following values:

`ECONNABORTED`: A connection has been aborted.
`EINTR`: The function was interrupted by a signal.
`EMFILE`: The maximum number of file descriptors in the process are already open.
`ENFILE`: The maximum number of file descriptors in the system are already open.
`ENOBUFS`: No buffer space is available.
`ENOMEM`: There was insufficient memory available to complete the operation.
`ETIMEDOUT`: Deadline was reached.

ipaddr tcpaddr(tcpsock s);

After accepting an incoming TCP connection this function can be used to retrieve the IP address of the peer.

The function cannot fail.

void tcpclose(tcpsock s);

Closes the socket.

The function always succeeds.

tcpsock tcpconnect(ipaddr addr, int64_t deadline);

Creates a TCP connection to a remote endpoint.

First parameter is the IP address to connect to. Second one is the `deadline` for the operation.

```
ipaddr addr = ipremote("www.example.org", 80, 0, -1);
tcpsock connection = tcpconnect(addr, -1);
```

If the function succeeds it returns handle of the newly created connection and sets `errno` to zero.

If the function fails it returns `NULL` and sets `errno` to one of the following values:

`ECONNREFUSED`: The target address was not listening for connections or refused the connection request.
`ECONNRESET`: Remote host reset the connection request.
`EHOSTUNREACH`: The destination host cannot be reached.
`ENETDOWN`: The local network interface used to reach the destination is down.
`ENETUNREACH`: No route to the network is present.
`EINTR`: The function was interrupted by a signal.
`EMFILE`: The maximum number of file descriptors in the process are already open.
`ENFILE`: The maximum number of file descriptors in the system are already open.
`EACCES`: The process does not have appropriate privileges.
`ENOBUFS`: No buffer space is available.
`ENOMEM`: There was insufficient memory available to complete the operation.
`ETIMEDOUT`: Deadline was reached.

void tcpflush(tcpsock s, int64_t deadline);

When using `tcpsend()` the library may choose not to send the data for the moment. `tcpflush()` can be used to flush any yet unsent data to the network.

First argument is the socket to flush, second one is the deadline.

In case of success, the function sets errno to zero. In case of failure it sets it to one of the following values:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EINTR`: The function was interrupted by a signal.
`EPIPE`: The socket is shut down for writing.
`EACCES`: The process does not have appropriate privileges.
`ENETDOWN`: The local network interface used to reach the destination is down.
`ENETUNREACH`: No route to the network is present.
`ENOBUFS`: No buffer space is available.

ETIMEDOUT: Deadline was reached.

tcpsock tcplisten(ipaddr addr, int backlog);

Starts listening for incoming TCP connections.

First argument is the IP address to listen on (see iplocal()), second one is the connection backlog size, i.e. maximum number of connections that weren't refused but are not yet accepted.

```
ipaddr addr = iplocal("eth0", 5555, 0);
tcpsock listener = tcplisten(addr, 10);
```

If the IP address passed to the function has port number set to zero, operating system will choose an ephemeral port to bind to. The number of the chosen port can be later retrieved using tcpport() function.

In case of success tcplisten() returns handle to the listening socket and sets errno to zero.

In case of failure it returns NULL and sets errno to one of the following error codes:

EADDRINUSE: The specified address is already in use.
EADDRNOTAVAIL: The specified address is not available from the local machine.
EMFILE: No more file descriptors are available for this process.
ENFILE: No more file descriptors are available for the system.
EACCES: The process does not have appropriate privileges.
ENOBUFS: Insufficient resources were available in the system to perform the operation.
ENOMEM: Insufficient memory was available to fulfill the request.

int tcpport(tcpsock s);

Returns TCP port used by the socket.

If tcplisten() was called with port number set to zero, operating system chooses an ephemeral port for the connection. This function can be used to retrieve the port number in that scenario.

The function cannot return error.

size_t tcprecv(tcpsock s, void *buf, size_t len, int64_t deadline);

Reads data from TCP connection.

First argument is the socket to read from. Second parameter is pointer to the buffer to read the data to. Third argument is the size of the buffer. Finally, fourth parameter is deadline for the operation.

```
char buf[256];
size_t nbytes = tcprecv(s, buf, sizeof(buf), -1);
```

In the case of success, errno is set to zero. In the case of failure it is set to one of the following values:

ECONNRESET: A connection was forcibly closed by a peer.
EINTR: The function was interrupted by a signal.
ENOBUFS: Insufficient resources were available in the system to perform the operation.
ENOMEM: Insufficient memory was available to fulfill the request.
ETIMEDOUT: Deadline exceeded.

The function returns number of bytes received.

size_t tcprecvuntil(tcpsock s, void *buf, size_t len, const char *delims, size_t delimcount, int64_t deadline);

Reads data from TCP connection until either buffer is full or a delimiter character is reached.

First argument is the socket to read from. Second parameter is pointer to the buffer to read the data to. Third argument is the size of the buffer.

`delims` parameter is a sequence of bytes any of which, if encountered in the incoming data, will cause `tcprecvuntil()` to exit. Size of the sequence, in bytes, is passed to the function in `delimcount` parameter.

Finally, last parameter is deadline for the operation.

For example, following example reads data until either comma or newline character is encountered:

```
char buf[256];
size_t nbytes = tcprecvuntil(s, buf, sizeof(buf), ",\n", 2, -1);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following values:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EINTR`: The function was interrupted by a signal.
`ENOBUFS`: Insufficient resources were available in the system to perform the operation.
`ENOMEM`: Insufficient memory was available to fulfill the request.
`ETIMEDOUT`: Deadline exceeded.

The function returns number of bytes received including the delimiter character.

size_t tcpsend(tcpsock s, const void *buf, size_t len, int64_t deadline);

Sends data to TCP connection.

First argument is the socket handle to send the data to. Second argument is pointer to the buffer to send. Third argument is the size of the buffer, in bytes. Finally, fourth argument is the `deadline`.

```
size_t nbytes = tcpsend(s, "ABC", 3, -1);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following error codes:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EINTR`: The function was interrupted by a signal.
`EPIPE`: The socket is shut down for writing.
`EACCES`: The process does not have appropriate privileges.
`ENETDOWN`: The local network interface used to reach the destination is down.
`ENETUNREACH`: No route to the network is present.
`ENOBUFS`: No buffer space is available.
`ETIMEDOUT`: Deadline was reached.

Whether successful or not, the function returns number of bytes actually written to the socket.

To optimise the throughput `tcpsend()` may store the data into output buffer and not flush it into the network immediately. To force sending of the data use `tcpflush()` function.

void udpclose(udpsock s);

Closes the socket.

The function always succeeds.

udpsock udplisten(ipaddr addr);

Creates an UDP socket and starts listening for incoming packets.

The argument is the IP address to listen on (see `iplocal()`).

```
ipaddr addr = iplocal("eth0", 5555, 0);
udpsock s = udplisten(addr);
```

If the IP address passed to the function has port number set to zero, operating system will choose an ephemeral port to bind to. The number of the chosen port can be later retrieved using `udpport()` function.

In case of success `udplisten()` returns handle to the listening socket and sets `errno` to zero.

In case of failure it returns `NULL` and sets `errno` to one of the following error codes:

`EADDRINUSE`: The specified address is already in use.
`EADDRNOTAVAIL`: The specified address is not available from the local machine.
`EMFILE`: No more file descriptors are available for this process.
`ENFILE`: No more file descriptors are available for the system.
`EACCES`: The process does not have appropriate privileges.
`ENOBUFS`: Insufficient resources were available in the system to perform the operation.
`ENOMEM`: Insufficient memory was available to fulfill the request.

int udpport(udpsock s);

Returns UDP port used by the socket.

If `udplisten()` was called with port number set to zero, operating system chooses an ephemeral port to listen on. `udpport()` function can be used to retrieve the port number in that scenario.

The function cannot return error.

size_t udprecv(udpsock s, ipaddr *addr, void *buf, size_t len, int64_t deadline);

Receives an UDP datagram.

First argument is the socket to receive from. Second parameter is an out parameter. If the call succeeds it will hold the source IP address of the datagram. Third argument is the buffer to read the data to. Fourth is the size of the buffer. Last parameter is deadline for the operation.

```
ipaddr addr;
char buf[256];
size_t nbytes = udprecv(s, &addr, buf, sizeof(buf), -1);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following values:

`EINTR`: The function was interrupted by a signal.
`ENOBUFS`: Insufficient resources were available in the system to perform the operation.
`ENOMEM`: Insufficient memory was available to fulfill the request.
`ETIMEDOUT`: Deadline exceeded.

The function returns number of bytes in the datagram or zero if no datagram was received.

void udpsend(udpsock s, ipaddr addr, const void *buf, size_t len);

Sends an UDP datagram.

First argument is the socket handle to send the data to. Second argument is the IP address to the datagram to. Third argument points to the data to send. Fourth argument is the size of the buffer, in bytes.

```
ipaddr addr = ipremote("192.168.0.111", 5555, 0, -1);
udpsend(s, addr, "Hello, world!", 13);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following error codes:

`EMSGSIZE`: The message is too large to be sent all at once.
`EINTR`: The function was interrupted by a signal.
`EACCES`: The process does not have appropriate privileges.
`ENETDOWN`: The local network interface used to reach the destination is down.
`ENETUNREACH`: No route to the network is present.
`ENOBUFS`: No buffer space is available.
`ETIMEDOUT`: Deadline was reached.

unixsock unixaccept(unixsock s, int64_t deadline);

Accepts new incoming UNIX domain connection from `listening` socket `s`. Second argument is the `deadline` for the operation.

```
unixsock listener = unixlisten("/tmp/test.unix", 10);
unixsock connection = unixaccept(listener, -1);
```

If the function succeeds it returns handle of the newly created connection and sets `errno` to zero.

If the function fails it returns `NULL` and sets `errno` to one of the following values:

`ECONNABORTED`: A connection has been aborted.
`EINTR`: The function was interrupted by a signal.
`EMFILE`: The maximum number of file descriptors in the process are already open.
`ENFILE`: The maximum number of file descriptors in the system are already open.
`ENOBUFS`: No buffer space is available.
`ENOMEM`: There was insufficient memory available to complete the operation.
`ETIMEDOUT`: Deadline was reached.

void unixclose(unixsock s);

Closes the socket.

The function always succeeds.

unixsock unixconnect(const char *addr);

Creates a UNIX domain connection to a remote endpoint.

The parameter is the file to connect to.

```
unixsock connection = unixconnect("/tmp/test.unix");
```

If the function succeeds it returns handle of the newly created connection and sets `errno` to zero.

If the function fails it returns `NULL` and sets `errno` to one of the following values:

`ECONNREFUSED`: The target address was not listening for connections or refused the connection request.
`ECONNRESET`: Remote host reset the connection request.
`EIO`: An I/O error occurred while reading from or writing to the file system.
`ELOOP`: A loop exists in symbolic links encountered during resolution of the pathname in address.
`ENAMETOOLONG`: A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX` characters.
`ENOENT`: A component of the pathname does not name an existing file or the pathname is an empty string.
`ENOTDIR`: A component of the path prefix of the pathname in address is not a directory.
`EINTR`: The function was interrupted by a signal.
`EMFILE`: The maximum number of file descriptors in the process are already open.
`ENFILE`: The maximum number of file descriptors in the system are already open.
`EACCES`: The process does not have appropriate privileges.
`ENOBUFS`: No buffer space is available.
`ENOMEM`: There was insufficient memory available to complete the operation.

void unixflush(unixsock s, int64_t deadline);

When using `unixsend()` the library may choose not to send the data for the moment. `unixflush()` can be used to flush any yet unsent data.

First argument is the socket to flush, second one is the deadline.

In case of success, the function sets errno to zero. In case of failure it sets it to one of the following values:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EINTR`: The function was interrupted by a signal.
`EPIPE`: The socket is shut down for writing.
`EACCES`: The process does not have appropriate privileges.
`EIO`: An I/O error occurred while reading from or writing to the file system.
`ENOBUFS`: No buffer space is available.
`ETIMEDOUT`: Deadline was reached.

unixsock unixlisten(const char *addr, int backlog);

Starts listening for incoming UNIX domain connections.

First argument is the file to listen on second one is the connection backlog size, i.e. maximum number of connections that weren't refused but are not yet accepted.

```
unixsock listener = unixlisten("/tmp/test.unix", 10);
```

In case of success `unixlisten()` returns handle to the listening socket and sets `errno` to zero.

In case of failure it returns `NULL` and sets `errno` to one of the following error codes:

`EDESTADDRREQ or EISDIR`: The address argument is a null pointer.
`EIO`: An I/O error occurred.
`ELOOP`: A loop exists in symbolic links encountered during resolution of the pathname in address.
`ENAMETOOLONG`: A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX` characters.
`ENOENT`: A component of the pathname does not name an existing file or the pathname is an empty string.
`ENOTDIR`: A component of the path prefix of the pathname in address is not a directory.
`EROFS`: The name would reside on a read-only file system.
`EADDRINUSE`: The specified address is already in use.
`EMFILE`: No more file descriptors are available for this process.
`ENFILE`: No more file descriptors are available for the system.
`EACCES`: The process does not have appropriate privileges.
`ENOBUFS`: Insufficient resources were available in the system to perform the operation.
`ENOMEM`: Insufficient memory was available to fulfill the request.

void unixpair(unixsock *a, unixsock *b);

Creates a pair of mutually connected UNIX domain sockets.

The arguments are pointers to two uninitialised `unixsock` handles.

If the function succeeds it sets `errno` to zero.

If it fails it sets it to one of the following error codes:

`EINVAL`: One or both arguments are `NULL`.
`EMFILE`: The maximum number of file descriptors in the process are already open.
`ENFILE`: The maximum number of file descriptors in the system are already open.
`EACCES`: The process does not have appropriate privileges.
`ENOBUFS`: No buffer space is available.
`ENOMEM`: There was insufficient memory available to complete the operation.

size_t unixrecv(unixsock s, void *buf, size_t len, int64_t deadline);

Reads data from UNIX domain connection.

First argument is the socket to read from. Second parameter is pointer to the buffer to read the data to. Third argument is the size of the buffer. Finally, fourth parameter is deadline for the operation.

```
char buf[256];
size_t nbytes = unixrecv(s, buf, sizeof(buf), -1);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following values:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EIO`: An I/O error occurred.
`EINTR`: The function was interrupted by a signal.
`ENOBUFS`: Insufficient resources were available in the system to perform the operation.
`ENOMEM`: Insufficient memory was available to fulfill the request.
`ETIMEDOUT`: Deadline exceeded.

The function returns number of bytes received.

size_t unixrecvuntil(unixsock s, void *buf, size_t len, const char *delims, size_t delimcount, int64_t deadline);

Reads data from UNIX domain connection until either buffer is full or a delimiter character is reached.

First argument is the socket to read from. Second parameter is pointer to the buffer to read the data to. Third argument is the size of the buffer.

`delims` parameter is a sequence of bytes any of which, if encountered in the incoming data, will cause `tcprecvuntil()` to exit. Size of the sequence, in bytes, is passed to the function in `delimcount` parameter.

Finally, last parameter is deadline for the operation.

For example, following example reads data until either comma or newline character is encountered:

```
char buf[256];
size_t nbytes = unixrecvuntil(s, buf, sizeof(buf), ",\n", 2, -1);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following values:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EIO`: An I/O error occurred.
`EINTR`: The function was interrupted by a signal.
`ENOBUFS`: Insufficient resources were available in the system to perform the operation.
`ENOMEM`: Insufficient memory was available to fulfill the request.
`ETIMEDOUT`: Deadline exceeded.

The function returns number of bytes received including the delimiter character.

size_t unixsend(unixsock s, const void *buf, size_t len, int64_t deadline);

Sends data to UNIX domain connection.

First argument is the socket handle to send the data to. Second argument is pointer to the buffer to send. Third argument is the size of the buffer, in bytes. Finally, fourth argument is the `deadline`.

```
size_t nbytes = unixsend(s, "ABC", 3, -1);
```

In the case of success, `errno` is set to zero. In the case of failure it is set to one of the following error codes:

`ECONNRESET`: A connection was forcibly closed by a peer.
`EINTR`: The function was interrupted by a signal.
`EPIPE`: The socket is shut down for writing.
`EACCES`: The process does not have appropriate privileges.
`EIO`: An I/O error occurred while reading from or writing to the file system.
`ENOBUFS`: No buffer space is available.
`ETIMEDOUT`: Deadline was reached.

Whether successful or not, the function returns number of bytes actually written to the socket.

To optimise the throughput `unixsend()` may store the data into output buffer and not flush it to the peer immediately. To force sending of the data use `unixflush()` function.

void yield(void);

By calling this function you give other coroutines a chance to run.

You should consider using `yield()` when doing lengthy computations which don't involve natural coroutine switching points such as network operations.

The function has no return value, nor can it return an error.