NAME

AnyEvent::Util – various utility functions.

SYNOPSIS

```
    use AnyEvent::Util;
```

DESCRIPTION

This module implements various utility functions, mostly replacing well-known functions by event-ised counterparts.

All functions documented without `AnyEvent::Util::` prefix are exported by default.

($r, `$w`) = portable_pipe

Calling `pipe` in Perl is portable – except it doesn't really work on sucky windows platforms (at least not with most perls – cygwin's perl notably works fine): On windows, you actually get two file handles you cannot use select on.

This function gives you a pipe that actually works even on the broken windows platform (by creating a pair of TCP sockets on windows, so do not expect any speed from that) and using `pipe` everywhere else.

See `portable_socketpair`, below, for a bidirectional "pipe".

Returns the empty list on any errors.

($fh1, `$fh2`) = portable_socketpair

Just like `portable_pipe`, above, but returns a bidirectional pipe (usually by calling `socketpair` to create a local loopback socket pair, except on windows, where it again returns two interconnected TCP sockets).

Returns the empty list on any errors.

fork_call { CODE } `@args`, `$cb`–>(@res)

Executes the given code block asynchronously, by forking. Everything the block returns will be transferred to the calling process (by serialising and deserialising via Storable).

If there are any errors, then the `$cb` will be called without any arguments. In that case, either `$@` contains the exception (and `$!` is irrelevant), or `$!` contains an error number. In all other cases, `$@` will be `undef`ined.

The code block must not ever call an event-polling function or use event-based programming that might cause any callbacks registered in the parent to run.

Win32 spoilers: Due to the endlessly sucky and broken native windows perls (there is no way to cleanly exit a child process on that platform that doesn't also kill the parent), you have to make sure that your main program doesn't exit as long as any `fork_calls` are still in progress, otherwise the program won't exit. Also, on most windows platforms some memory will leak for every invocation. We are open for improvements that don't require XS hackery.

Note that forking can be expensive in large programs (RSS 200MB+). On windows, it is abysmally slow, do not expect more than 5..20 forks/s on that sucky platform (note this uses perl's pseudo-threads, so avoid those like the plague).

Example: poor man's async disk I/O (better use AnyEvent::IO together with IO::AIO).

```
    fork_call {
       open my $fh, "</etc/passwd"
          or die "passwd: $!";
       local $/;
       <$fh>
    } sub {
       my ($passwd) = @_;
       ...
    };
```

`$AnyEvent::Util::MAX_FORKS` [default: 10]
: The maximum number of child processes that `fork_call` will fork in parallel. Any additional requests will be queued until a slot becomes free again.

  The environment variable `PERL_ANYEVENT_MAX_FORKS` is used to initialise this value.

fh_nonblocking `$fh`, `$nonblocking`
: Sets the blocking state of the given filehandle (true == nonblocking, false == blocking). Uses fcntl on anything sensible and ioctl FIONBIO on broken (i.e. windows) platforms.

  Instead of using this function, you could use `AnyEvent::fh_block` or `AnyEvent::fh_unblock`.

`$guard` = guard { CODE }
: This function creates a special object that, when destroyed, will execute the code block.

  This is often handy in continuation-passing style code to clean up some resource regardless of where you break out of a process.

  The Guard module will be used to implement this function, if it is available. Otherwise a pure-perl implementation is used.

  While the code is allowed to throw exceptions in unusual conditions, it is not defined whether this exception will be reported (at the moment, the Guard module and AnyEvent's pure-perl implementation both try to report the error and continue).

  You can call one method on the returned object:

`$guard`−>cancel
: This simply causes the code block not to be invoked: it "cancels" the guard.

AnyEvent::Util::close_all_fds_except `@fds`
: This rarely-used function simply closes all file descriptors (or tries to) of the current process except the ones given as arguments.

  When you want to start a long-running background server, then it is often beneficial to do this, as too many C−libraries are too stupid to mark their internal fd's as close-on-exec.

  The function expects to be called shortly before an `exec` call.

  Example: close all fds except 0, 1, 2.

```
       close_all_fds_except 0, 2, 1;
```

`$cv` = run_cmd `$cmd`, key => value...
: Run a given external command, potentially redirecting file descriptors and return a condition variable that gets sent the exit status (like `$?`) when the program exits *and* all redirected file descriptors have been exhausted.

  The `$cmd` is either a single string, which is then passed to a shell, or an arrayref, which is passed to the `execvp` function (the first array element is used both for the executable name and argv[0]).

  The key-value pairs can be:

">" => `$filename`
: Redirects program standard output into the specified filename, similar to `>filename` in the shell.

">" => \$data
: Appends program standard output to the referenced scalar. The condvar will not be signalled before EOF or an error is signalled.

  Specifying the same scalar in multiple ">" pairs is allowed, e.g. to redirect both stdout and stderr into the same scalar:

```
   ">"  => \$output,
   "2>" => \$output,
```

">" => `$filehandle`
: Redirects program standard output to the given filehandle (or actually its underlying file descriptor).

">" => `$callback->`($data)
: Calls the given callback each time standard output receives some data, passing it the data received. On EOF or error, the callback will be invoked once without any arguments.

  The condvar will not be signalled before EOF or an error is signalled.

"fd>" => `$see_above`
: Like ">", but redirects the specified fd number instead.

"<" => `$see_above`
: The same, but redirects the program's standard input instead. The same forms as for ">" are allowed.

  In the callback form, the callback is supposed to return data to be written, or the empty list or `undef` or a zero-length scalar to signal EOF.

  Similarly, either the write data must be exhausted or an error is to be signalled before the condvar is signalled, for both string-reference and callback forms.

"fd<" => `$see_above`
: Like "<", but redirects the specified file descriptor instead.

on_prepare => `$cb`
: Specify a callback that is executed just before the command is `exec`'ed, in the child process. Be careful not to use any event handling or other services not available in the child.

  This can be useful to set up the environment in special ways, such as changing the priority of the command or manipulating signal handlers (e.g. setting `SIGINT` to `IGNORE`).

close_all => `$boolean`
: When `close_all` is enabled (default is disabled), then all extra file descriptors will be closed, except the ones that were redirected and `0`, `1` and `2`.

  See `close_all_fds_except` for more details.

'$$' => \$pid
: A reference to a scalar which will receive the PID of the newly-created subprocess after `run_cmd` returns.

  Note the the PID might already have been recycled and used by an unrelated process at the time `run_cmd` returns, so it's not useful to send signals, use as a unique key in data structures and so on.

Example: run `rm -rf /`, redirecting standard input, output and error to */dev/null*.

```
   my $cv = run_cmd [qw(rm -rf /)],
      "<", "/dev/null",
      ">", "/dev/null",
      "2>", "/dev/null";
   $cv->recv and die "d'oh! something survived!"
```

Example: run *openssl* and create a self-signed certificate and key, storing them in `$cert` and `$key`. When finished, check the exit status in the callback and print key and certificate.

```
   my $cv = run_cmd [qw(openssl req
                     -new -nodes -x509 -days 3650
                     -newkey rsa:2048 -keyout /dev/fd/3
                     -batch -subj /CN=AnyEvent
                    )],
      "<", "/dev/null",
      ">" , \my $cert,
      "3>", \my $key,
      "2>", "/dev/null";

   $cv->cb (sub {
      shift->recv and die "openssl failed";

      print "$key\n$cert\n";
   });
```

AnyEvent::Util::punycode_encode `$string`

Punycode-encodes the given `$string` and returns its punycode form. Note that uppercase letters are *not* casefolded – you have to do that yourself.

Croaks when it cannot encode the string.

AnyEvent::Util::punycode_decode `$string`

Tries to punycode-decode the given `$string` and return its unicode form. Again, uppercase letters are not casefoled, you have to do that yourself.

Croaks when it cannot decode the string.

AnyEvent::Util::idn_nameprep `$idn`[, `$display`]

Implements the IDNA nameprep normalisation algorithm. Or actually the UTS#46 algorithm. Or maybe something similar – reality is complicated between IDNA2003, UTS#46 and IDNA2008. If `$display` is true then the name is prepared for display, otherwise it is prepared for lookup (default).

If you have no clue what this means, look at `idn_to_ascii` instead.

This function is designed to avoid using a lot of resources – it uses about 1MB of RAM (most of this due to Unicode::Normalize). Also, names that are already "simple" will only be checked for basic validity, without the overhead of full nameprep processing.

`$domainname` = AnyEvent::Util::idn_to_ascii `$idn`

Converts the given unicode string (`$idn`, international domain name, e.g.) to a pure-ASCII domain name (this is usually called the "IDN ToAscii" transform). This transformation is idempotent, which means you can call it just in case and it will do the right thing.

Unlike some other "ToAscii" implementations, this one works on full domain names and should never fail – if it cannot convert the name, then it will return it unchanged.

This function is an amalgam of IDNA2003, UTS#46 and IDNA2008 – it tries to be reasonably compatible to other implementations, reasonably secure, as much as IDNs can be secure, and reasonably efficient when confronted with IDNs that are already valid DNS names.

`$idn` = AnyEvent::Util::idn_to_unicode `$idn`

Converts the given unicode string (`$idn`, international domain name, e.g. , www.deliantra.net, www.xn—l–0ga.de) to unicode form (this is usually called the "IDN ToUnicode" transform). This transformation is idempotent, which means you can call it just in case and it will do the right thing.

Unlike some other "ToUnicode" implementations, this one works on full domain names and should never fail – if it cannot convert the name, then it will return it unchanged.

This function is an amalgam of IDNA2003, UTS#46 and IDNA2008 – it tries to be reasonably compatible to other implementations, reasonably secure, as much as IDNs can be secure, and reasonably efficient when confronted with IDNs that are already valid DNS names.

At the moment, this function simply calls `idn_nameprep $idn, 1`, returning its argument when that function fails.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```