NAME

AnyEvent – the DBI of event loop programming

EV, Event, Glib, Tk, UV, Perl, Event::Lib, Irssi, rxvt–unicode, IO::Async, Qt, FLTK and POE are various supported event loops/environments.

SYNOPSIS

```
use AnyEvent;

# if you prefer function calls, look at the AE manpage for
# an alternative API.

# file handle or descriptor readable
my $w = AnyEvent->io (fh => $fh, poll => "r", cb => sub { ...  });

# one-shot or repeating timers
my $w = AnyEvent->timer (after => $seconds, cb => sub { ...  });
my $w = AnyEvent->timer (after => $seconds, interval => $seconds, cb => ...);

print AnyEvent->now;  # prints current event loop time
print AnyEvent->time; # think Time::HiRes::time or simply CORE::time.

# POSIX signal
my $w = AnyEvent->signal (signal => "TERM", cb => sub { ... });

# child process exit
my $w = AnyEvent->child (pid => $pid, cb => sub {
   my ($pid, $status) = @_;
   ...
});

# called when event loop idle (if applicable)
my $w = AnyEvent->idle (cb => sub { ... });

my $w = AnyEvent->condvar; # stores whether a condition was flagged
$w->send; # wake up current and all future recv's
$w->recv; # enters "main loop" till $condvar gets ->send
# use a condvar in callback mode:
$w->cb (sub { $_[0]->recv });
```

INTRODUCTION/TUTORIAL

This manpage is mainly a reference manual. If you are interested in a tutorial or some gentle introduction, have a look at the AnyEvent::Intro manpage.

SUPPORT

An FAQ document is available as AnyEvent::FAQ.

There also is a mailinglist for discussing all things AnyEvent, and an IRC channel, too.

See the AnyEvent project page at the **Schmorpforge Ta-Sa Software Repository**, at <http://anyevent.schmorp.de>, for more info.

WHY YOU SHOULD USE THIS MODULE (OR NOT)

Glib, POE, IO::Async, Event... CPAN offers event models by the dozen nowadays. So what is different about AnyEvent?

Executive Summary: AnyEvent is *compatible*, AnyEvent is *free of policy* and AnyEvent is *small and efficient*.

First and foremost, *AnyEvent is not an event model* itself, it only interfaces to whatever event model the main program happens to use, in a pragmatic way. For event models and certain classes of immortals alike, the statement "there can only be one" is a bitter reality: In general, only one event loop can be active at the same time in a process. AnyEvent cannot change this, but it can hide the differences between those event loops.

The goal of AnyEvent is to offer module authors the ability to do event programming (waiting for I/O or timer events) without subscribing to a religion, a way of living, and most importantly: without forcing your module users into the same thing by forcing them to use the same event model you use.

For modules like POE or IO::Async (which is a total misnomer as it is actually doing all I/O *synchronously*...), using them in your module is like joining a cult: After you join, you are dependent on them and you cannot use anything else, as they are simply incompatible to everything that isn't them. What's worse, all the potential users of your module are *also* forced to use the same event loop you use.

AnyEvent is different: AnyEvent + POE works fine. AnyEvent + Glib works fine. AnyEvent + Tk works fine etc. etc. but none of these work together with the rest: POE + EV? No go. Tk + Event? No go. Again: if your module uses one of those, every user of your module has to use it, too. But if your module uses AnyEvent, it works transparently with all event models it supports (including stuff like IO::Async, as long as those use one of the supported event loops. It is easy to add new event loops to AnyEvent, too, so it is future-proof).

In addition to being free of having to use *the one and only true event model*, AnyEvent also is free of bloat and policy: with POE or similar modules, you get an enormous amount of code and strict rules you have to follow. AnyEvent, on the other hand, is lean and to the point, by only offering the functionality that is necessary, in as thin as a wrapper as technically possible.

Of course, AnyEvent comes with a big (and fully optional!) toolbox of useful functionality, such as an asynchronous DNS resolver, 100% non-blocking connects (even with TLS/SSL, IPv6 and on broken platforms such as Windows) and lots of real-world knowledge and workarounds for platform bugs and differences.

Now, if you *do want* lots of policy (this can arguably be somewhat useful) and you want to force your users to use the one and only event model, you should *not* use this module.

DESCRIPTION

AnyEvent provides a uniform interface to various event loops. This allows module authors to use event loop functionality without forcing module users to use a specific event loop implementation (since more than one event loop cannot coexist peacefully).

The interface itself is vaguely similar, but not identical to the Event module.

During the first call of any watcher-creation method, the module tries to detect the currently loaded event loop by probing whether one of the following modules is already loaded: EV, AnyEvent::Loop, Event, Glib, Tk, Event::Lib, Qt, POE. The first one found is used. If none are detected, the module tries to load the first four modules in the order given; but note that if EV is not available, the pure-perl AnyEvent::Loop should always work, so the other two are not normally tried.

Because AnyEvent first checks for modules that are already loaded, loading an event model explicitly before first using AnyEvent will likely make that model the default. For example:

```
use Tk;
use AnyEvent;

# .. AnyEvent will likely default to Tk
```

The *likely* means that, if any module loads another event model and starts using it, all bets are off – this case should be very rare though, as very few modules hardcode event loops without announcing this very loudly.

The pure-perl implementation of AnyEvent is called `AnyEvent::Loop`. Like other event modules you can load it explicitly and enjoy the high availability of that event loop :)

WATCHERS

AnyEvent has the central concept of a *watcher*, which is an object that stores relevant data for each kind of event you are waiting for, such as the callback to call, the file handle to watch, etc.

These watchers are normal Perl objects with normal Perl lifetime. After creating a watcher it will immediately "watch" for events and invoke the callback when the event occurs (of course, only when the event model is in control).

Note that **callbacks must not permanently change global variables** potentially in use by the event loop (such as `$_` or `$[`) and that **callbacks must not `die`**. The former is good programming practice in Perl and the latter stems from the fact that exception handling differs widely between event loops.

To disable a watcher you have to destroy it (e.g. by setting the variable you store it in to `undef` or otherwise deleting all references to it).

All watchers are created by calling a method on the `AnyEvent` class.

Many watchers either are used with "recursion" (repeating timers for example), or need to refer to their watcher object in other ways.

One way to achieve that is this pattern:

```
my $w; $w = AnyEvent->type (arg => value ..., cb => sub {
   # you can use $w here, for example to undef it
   undef $w;
});
```

Note that `my $w; $w =` combination. This is necessary because in Perl, my variables are only visible after the statement in which they are declared.

I/O WATCHERS

```
$w = AnyEvent->io (
   fh   => <filehandle_or_fileno>,
   poll => <"r" or "w">,
   cb   => <callback>,
);
```

You can create an I/O watcher by calling the `AnyEvent->io` method with the following mandatory key-value pairs as arguments:

`fh` is the Perl *file handle* (or a naked file descriptor) to watch for events (AnyEvent might or might not keep a reference to this file handle). Note that only file handles pointing to things for which non-blocking operation makes sense are allowed. This includes sockets, most character devices, pipes, fifos and so on, but not for example files or block devices.

`poll` must be a string that is either `r` or `w`, which creates a watcher waiting for "r"eadable or "w"ritable events, respectively.

`cb` is the callback to invoke each time the file handle becomes ready.

Although the callback might get passed parameters, their value and presence is undefined and you cannot rely on them. Portable AnyEvent callbacks cannot use arguments passed to I/O watcher callbacks.

The I/O watcher might use the underlying file descriptor or a copy of it. You must not close a file handle as long as any watcher is active on the underlying file descriptor.

Some event loops issue spurious readiness notifications, so you should always use non-blocking calls when reading/writing from/to your file handles.

Example: wait for readability of STDIN, then read a line and disable the watcher.

```
my $w; $w = AnyEvent->io (fh => \*STDIN, poll => 'r', cb => sub {
   chomp (my $input = <STDIN>);
   warn "read: $input\n";
   undef $w;
});
```

TIME WATCHERS

```
$w = AnyEvent->timer (after => <seconds>, cb => <callback>);

$w = AnyEvent->timer (
   after    => <fractional_seconds>,
   interval => <fractional_seconds>,
   cb       => <callback>,
);
```

You can create a time watcher by calling the `AnyEvent->timer` method with the following mandatory arguments:

`after` specifies after how many seconds (fractional values are supported) the callback should be invoked. `cb` is the callback to invoke in that case.

Although the callback might get passed parameters, their value and presence is undefined and you cannot rely on them. Portable AnyEvent callbacks cannot use arguments passed to time watcher callbacks.

The callback will normally be invoked only once. If you specify another parameter, `interval`, as a strictly positive number (> 0), then the callback will be invoked regularly at that interval (in fractional seconds) after the first invocation. If `interval` is specified with a false value, then it is treated as if it were not specified at all.

The callback will be rescheduled before invoking the callback, but no attempt is made to avoid timer drift in most backends, so the interval is only approximate.

Example: fire an event after 7.7 seconds.

```
my $w = AnyEvent->timer (after => 7.7, cb => sub {
   warn "timeout\n";
});

# to cancel the timer:
undef $w;
```

Example 2: fire an event after 0.5 seconds, then roughly every second.

```
my $w = AnyEvent->timer (after => 0.5, interval => 1, cb => sub {
   warn "timeout\n";
});
```

TIMING ISSUES

There are two ways to handle timers: based on real time (relative, “fire in 10 seconds”) and based on wallclock time (absolute, “fire at 12 o’clock”).

While most event loops expect timers to specified in a relative way, they use absolute time internally. This makes a difference when your clock “jumps”, for example, when ntp decides to set your clock backwards from the wrong date of 2014–01–01 to 2008–01–01, a watcher that is supposed to fire “after a second” might actually take six years to finally fire.

AnyEvent cannot compensate for this. The only event loop that is conscious of these issues is EV, which offers both relative (ev_timer, based on true relative time) and absolute (ev_periodic, based on wallclock time) timers.

AnyEvent always prefers relative timers, if available, matching the AnyEvent API.

AnyEvent has two additional methods that return the “current time”:

AnyEvent−>time

This returns the "current wallclock time" as a fractional number of seconds since the Epoch (the same thing as `time` or `Time::HiRes::time` return, and the result is guaranteed to be compatible with those).

It progresses independently of any event loop processing, i.e. each call will check the system clock, which usually gets updated frequently.

AnyEvent−>now

This also returns the "current wallclock time", but unlike `time`, above, this value might change only once per event loop iteration, depending on the event loop (most return the same time as `time`, above). This is the time that AnyEvent's timers get scheduled against.

In almost all cases (in all cases if you don't care), this is the function to call when you want to know the current time.

This function is also often faster then `AnyEvent->time`, and thus the preferred method if you want some timestamp (for example, AnyEvent::Handle uses this to update its activity timeouts).

The rest of this section is only of relevance if you try to be very exact with your timing; you can skip it without a bad conscience.

For a practical example of when these times differ, consider Event::Lib and EV and the following set-up:

The event loop is running and has just invoked one of your callbacks at time=500 (assume no other callbacks delay processing). In your callback, you wait a second by executing `sleep 1` (blocking the process for a second) and then (at time=501) you create a relative timer that fires after three seconds.

With Event::Lib, `AnyEvent->time` and `AnyEvent->now` will both return `501`, because that is the current time, and the timer will be scheduled to fire at time=504 (`501` + `3`).

With EV, `AnyEvent->time` returns `501` (as that is the current time), but `AnyEvent->now` returns `500`, as that is the time the last event processing phase started. With EV, your timer gets scheduled to run at time=503 (`500` + `3`).

In one sense, Event::Lib is more exact, as it uses the current time regardless of any delays introduced by event processing. However, most callbacks do not expect large delays in processing, so this causes a higher drift (and a lot more system calls to get the current time).

In another sense, EV is more exact, as your timer will be scheduled at the same time, regardless of how long event processing actually took.

In either case, if you care (and in most cases, you don't), then you can get whatever behaviour you want with any event loop, by taking the difference between `AnyEvent->time` and `AnyEvent->now` into account.

AnyEvent−>now_update

Some event loops (such as EV or AnyEvent::Loop) cache the current time for each loop iteration (see the discussion of AnyEvent−>now, above).

When a callback runs for a long time (or when the process sleeps), then this "current" time will differ substantially from the real time, which might affect timers and time-outs.

When this is the case, you can call this method, which will update the event loop's idea of "current time".

A typical example would be a script in a web server (e.g. `mod_perl`) − when mod_perl executes the script, then the event loop will have the wrong idea about the "current time" (being potentially far in the past, when the script ran the last time). In that case you should arrange a call to `AnyEvent->now_update` each time the web server process wakes up again (e.g. at the start of your script, or in a handler).

Note that updating the time *might* cause some events to be handled.

SIGNAL WATCHERS

```
$w = AnyEvent->signal (signal => <uppercase_signal_name>, cb => <callback>);
```

You can watch for signals using a signal watcher, `signal` is the signal *name* in uppercase and without any `SIG` prefix, `cb` is the Perl callback to be invoked whenever a signal occurs.

Although the callback might get passed parameters, their value and presence is undefined and you cannot rely on them. Portable AnyEvent callbacks cannot use arguments passed to signal watcher callbacks.

Multiple signal occurrences can be clumped together into one callback invocation, and callback invocation will be synchronous. Synchronous means that it might take a while until the signal gets handled by the process, but it is guaranteed not to interrupt any other callbacks.

The main advantage of using these watchers is that you can share a signal between multiple watchers, and AnyEvent will ensure that signals will not interrupt your program at bad times.

This watcher might use `%SIG` (depending on the event loop used), so programs overwriting those signals directly will likely not work correctly.

Example: exit on SIGINT

```
my $w = AnyEvent->signal (signal => "INT", cb => sub { exit 1 });
```

Restart Behaviour

While restart behaviour is up to the event loop implementation, most will not restart syscalls (that includes Async::Interrupt and AnyEvent's pure perl implementation).

Safe/Unsafe Signals

Perl signals can be either "safe" (synchronous to opcode handling) or "unsafe" (asynchronous) − the former might delay signal delivery indefinitely, the latter might corrupt your memory.

AnyEvent signal handlers are, in addition, synchronous to the event loop, i.e. they will not interrupt your running perl program but will only be called as part of the normal event handling (just like timer, I/O etc. callbacks, too).

Signal Races, Delays and Workarounds

Many event loops (e.g. Glib, Tk, Qt, IO::Async) do not support attaching callbacks to signals in a generic way, which is a pity, as you cannot do race-free signal handling in perl, requiring C libraries for this. AnyEvent will try to do its best, which means in some cases, signals will be delayed. The maximum time a signal might be delayed is 10 seconds by default, but can be overridden via `$ENV{PERL_ANYEVENT_MAX_SIGNAL_LATENCY}` or `$AnyEvent::MAX_SIGNAL_LATENCY` − see the "ENVIRONMENT VARIABLES" section for details.

All these problems can be avoided by installing the optional Async::Interrupt module, which works with most event loops. It will not work with inherently broken event loops such as Event or Event::Lib (and not with POE currently). For those, you just have to suffer the delays.

CHILD PROCESS WATCHERS

```
$w = AnyEvent->child (pid => <process id>, cb => <callback>);
```

You can also watch for a child process exit and catch its exit status.

The child process is specified by the `pid` argument (on some backends, using `0` watches for any child process exit, on others this will croak). The watcher will be triggered only when the child process has finished and an exit status is available, not on any trace events (stopped/continued).

The callback will be called with the pid and exit status (as returned by waitpid), so unlike other watcher types, you *can* rely on child watcher callback arguments.

This watcher type works by installing a signal handler for `SIGCHLD`, and since it cannot be shared, nothing else should use SIGCHLD or reap random child processes (waiting for specific child processes, e.g. inside `system`, is just fine).

There is a slight catch to child watchers, however: you usually start them *after* the child process was created, and this means the process could have exited already (and no SIGCHLD will be sent anymore).

Not all event models handle this correctly (neither POE nor IO::Async do, see their AnyEvent::Impl manpages for details), but even for event models that *do* handle this correctly, they usually need to be loaded before the process exits (i.e. before you fork in the first place). AnyEvent's pure perl event loop handles all cases correctly regardless of when you start the watcher.

This means you cannot create a child watcher as the very first thing in an AnyEvent program, you *have* to create at least one watcher before you `fork` the child (alternatively, you can call `AnyEvent::detect`).

As most event loops do not support waiting for child events, they will be emulated by AnyEvent in most cases, in which case the latency and race problems mentioned in the description of signal watchers apply.

Example: fork a process and wait for it

```
my $done = AnyEvent->condvar;

# this forks and immediately calls exit in the child. this
# normally has all sorts of bad consequences for your parent,
# so take this as an example only. always fork and exec,
# or call POSIX::_exit, in real code.
my $pid = fork or exit 5;

my $w = AnyEvent->child (
   pid => $pid,
   cb  => sub {
      my ($pid, $status) = @_;
      warn "pid $pid exited with status $status";
      $done->send;
   },
);

# do something else, then wait for process exit
$done->recv;
```

IDLE WATCHERS

```
$w = AnyEvent->idle (cb => <callback>);
```

This will repeatedly invoke the callback after the process becomes idle, until either the watcher is destroyed or new events have been detected.

Idle watchers are useful when there is a need to do something, but it is not so important (or wise) to do it instantly. The callback will be invoked only when there is "nothing better to do", which is usually defined as "all outstanding events have been handled and no new events have been detected". That means that idle watchers ideally get invoked when the event loop has just polled for new events but none have been detected. Instead of blocking to wait for more events, the idle watchers will be invoked.

Unfortunately, most event loops do not really support idle watchers (only EV, Event and Glib do it in a usable fashion) – for the rest, AnyEvent will simply call the callback "from time to time".

Example: read lines from STDIN, but only process them when the program is otherwise idle:

```
my @lines; # read data
my $idle_w;
my $io_w = AnyEvent->io (fh => \*STDIN, poll => 'r', cb => sub {
   push @lines, scalar <STDIN>;

   # start an idle watcher, if not already done
   $idle_w ||= AnyEvent->idle (cb => sub {
      # handle only one line, when there are lines left
```

```
         if (my $line = shift @lines) {
            print "handled when idle: $line";
         } else {
            # otherwise disable the idle watcher again
            undef $idle_w;
         }
      });
   });
```

CONDITION VARIABLES

```
   $cv = AnyEvent->condvar;

   $cv->send (<list>);
   my @res = $cv->recv;
```

If you are familiar with some event loops you will know that all of them require you to run some blocking "loop", "run" or similar function that will actively watch for new events and call your callbacks.

AnyEvent is slightly different: it expects somebody else to run the event loop and will only block when necessary (usually when told by the user).

The tool to do that is called a "condition variable", so called because they represent a condition that must become true.

Now is probably a good time to look at the examples further below.

Condition variables can be created by calling the `AnyEvent->condvar` method, usually without arguments. The only argument pair allowed is `cb`, which specifies a callback to be called when the condition variable becomes true, with the condition variable as the first argument (but not the results).

After creation, the condition variable is "false" until it becomes "true" by calling the `send` method (or calling the condition variable as if it were a callback, read about the caveats in the description for the `->send` method).

Since condition variables are the most complex part of the AnyEvent API, here are some different mental models of what they are – pick the ones you can connect to:

- Condition variables are like callbacks – you can call them (and pass them instead of callbacks). Unlike callbacks however, you can also wait for them to be called.
- Condition variables are signals – one side can emit or send them, the other side can wait for them, or install a handler that is called when the signal fires.
- Condition variables are like "Merge Points" – points in your program where you merge multiple independent results/control flows into one.
- Condition variables represent a transaction – functions that start some kind of transaction can return them, leaving the caller the choice between waiting in a blocking fashion, or setting a callback.
- Condition variables represent future values, or promises to deliver some result, long before the result is available.

Condition variables are very useful to signal that something has finished, for example, if you write a module that does asynchronous http requests, then a condition variable would be the ideal candidate to signal the availability of results. The user can either act when the callback is called or can synchronously `->recv` for the results.

You can also use them to simulate traditional event loops – for example, you can block your main program until an event occurs – for example, you could `->recv` in your main program until the user clicks the Quit button of your app, which would `->send` the "quit" event.

Note that condition variables recurse into the event loop – if you have two pieces of code that call `->recv` in a round-robin fashion, you lose. Therefore, condition variables are good to export to your caller, but you should avoid making a blocking wait yourself, at least in callbacks, as this asks for trouble.

Condition variables are represented by hash refs in perl, and the keys used by AnyEvent itself are all named `_ae_XXX` to make subclassing easy (it is often useful to build your own transaction class on top of AnyEvent). To subclass, use `AnyEvent::CondVar` as base class and call its `new` method in your own `new` method.

There are two "sides" to a condition variable – the "producer side" which eventually calls `-> send`, and the "consumer side", which waits for the send to occur.

Example: wait for a timer.

```
# condition: "wait till the timer is fired"
my $timer_fired = AnyEvent->condvar;

# create the timer - we could wait for, say
# a handle becomign ready, or even an
# AnyEvent::HTTP request to finish, but
# in this case, we simply use a timer:
my $w = AnyEvent->timer (
   after => 1,
   cb    => sub { $timer_fired->send },
);

# this "blocks" (while handling events) till the callback
# calls ->send
$timer_fired->recv;
```

Example: wait for a timer, but take advantage of the fact that condition variables are also callable directly.

```
my $done = AnyEvent->condvar;
my $delay = AnyEvent->timer (after => 5, cb => $done);
$done->recv;
```

Example: Imagine an API that returns a condvar and doesn't support callbacks. This is how you make a synchronous call, for example from the main program:

```
use AnyEvent::CouchDB;

...

my @info = $couchdb->info->recv;
```

And this is how you would just set a callback to be called whenever the results are available:

```
$couchdb->info->cb (sub {
   my @info = $_[0]->recv;
});
```

METHODS FOR PRODUCERS

These methods should only be used by the producing side, i.e. the code/module that eventually sends the signal. Note that it is also the producer side which creates the condvar in most cases, but it isn't uncommon for the consumer to create it as well.

`$cv`–>send (...)

Flag the condition as ready – a running `->recv` and all further calls to `recv` will (eventually) return after this method has been called. If nobody is waiting the send will be remembered.

If a callback has been set on the condition variable, it is called immediately from within send.

Any arguments passed to the `send` call will be returned by all future `->recv` calls.

Condition variables are overloaded so one can call them directly (as if they were a code reference). Calling them directly is the same as calling `send`.

`$cv`−>croak ($error)

Similar to send, but causes all calls to `−>recv` to invoke `Carp::croak` with the given error message/object/scalar.

This can be used to signal any errors to the condition variable user/consumer. Doing it this way instead of calling `croak` directly delays the error detection, but has the overwhelming advantage that it diagnoses the error at the place where the result is expected, and not deep in some event callback with no connection to the actual code causing the problem.

`$cv`−>begin ([group callback])
`$cv`−>end

These two methods can be used to combine many transactions/events into one. For example, a function that pings many hosts in parallel might want to use a condition variable for the whole process.

Every call to `−>begin` will increment a counter, and every call to `−>end` will decrement it. If the counter reaches `0` in `−>end`, the (last) callback passed to `begin` will be executed, passing the condvar as first argument. That callback is *supposed* to call `−>send`, but that is not required. If no group callback was set, `send` will be called without any arguments.

You can think of `$cv−>send` giving you an OR condition (one call sends), while `$cv−>begin` and `$cv−>end` giving you an AND condition (all `begin` calls must be `end`'ed before the condvar sends).

Let's start with a simple example: you have two I/O watchers (for example, STDOUT and STDERR for a program), and you want to wait for both streams to close before activating a condvar:

```
my $cv = AnyEvent->condvar;

$cv->begin; # first watcher
my $w1 = AnyEvent->io (fh => $fh1, cb => sub {
   defined sysread $fh1, my $buf, 4096
      or $cv->end;
});

$cv->begin; # second watcher
my $w2 = AnyEvent->io (fh => $fh2, cb => sub {
   defined sysread $fh2, my $buf, 4096
      or $cv->end;
});

$cv->recv;
```

This works because for every event source (EOF on file handle), there is one call to `begin`, so the condvar waits for all calls to `end` before sending.

The ping example mentioned above is slightly more complicated, as the there are results to be passed back, and the number of tasks that are begun can potentially be zero:

```
my $cv = AnyEvent->condvar;

my %result;
$cv->begin (sub { shift->send (\%result) });

for my $host (@list_of_hosts) {
   $cv->begin;
   ping_host_then_call_callback $host, sub {
      $result{$host} = ...;
      $cv->end;
   };
}
```

```
   $cv->end;

   ...

   my $results = $cv->recv;
```

This code fragment supposedly pings a number of hosts and calls `send` after results for all then have have been gathered – in any order. To achieve this, the code issues a call to `begin` when it starts each ping request and calls `end` when it has received some result for it. Since `begin` and `end` only maintain a counter, the order in which results arrive is not relevant.

There is an additional bracketing call to `begin` and `end` outside the loop, which serves two important purposes: first, it sets the callback to be called once the counter reaches `0`, and second, it ensures that `send` is called even when `no` hosts are being pinged (the loop doesn't execute once).

This is the general pattern when you "fan out" into multiple (but potentially zero) subrequests: use an outer `begin`/`end` pair to set the callback and ensure `end` is called at least once, and then, for each subrequest you start, call `begin` and for each subrequest you finish, call `end`.

METHODS FOR CONSUMERS

These methods should only be used by the consuming side, i.e. the code awaits the condition.

`$cv`–>recv

Wait (blocking if necessary) until the `->send` or `->croak` methods have been called on `$cv`, while servicing other watchers normally.

You can only wait once on a condition – additional calls are valid but will return immediately.

If an error condition has been set by calling `->croak`, then this function will call `croak`.

In list context, all parameters passed to `send` will be returned, in scalar context only the first one will be returned.

Note that doing a blocking wait in a callback is not supported by any event loop, that is, recursive invocation of a blocking `->recv` is not allowed and the `recv` call will `croak` if such a condition is detected. This requirement can be dropped by relying on Coro::AnyEvent , which allows you to do a blocking `->recv` from any thread that doesn't run the event loop itself. Coro::AnyEvent is loaded automatically when Coro is used with AnyEvent, so code does not need to do anything special to take advantage of that: any code that would normally block your program because it calls `recv`, be executed in an `async` thread instead without blocking other threads.

Not all event models support a blocking wait – some die in that case (programs might want to do that to stay interactive), so *if you are using this from a module, never require a blocking wait.* Instead, let the caller decide whether the call will block or not (for example, by coupling condition variables with some kind of request results and supporting callbacks so the caller knows that getting the result will not block, while still supporting blocking waits if the caller so desires).

You can ensure that `->recv` never blocks by setting a callback and only calling `->recv` from within that callback (or at a later time). This will work even when the event loop does not support blocking waits otherwise.

`$bool` = `$cv`–>ready

Returns true when the condition is "true", i.e. whether `send` or `croak` have been called.

`$cb` = `$cv`–>cb ($cb–>($cv))

This is a mutator function that returns the callback set (or `undef` if not) and optionally replaces it before doing so.

The callback will be called when the condition becomes "true", i.e. when `send` or `croak` are called, with the only argument being the condition variable itself. If the condition is already true, the callback is called immediately when it is set. Calling `recv` inside the callback or at any later time is guaranteed not to block.

Additionally, when the callback is invoked, it is also removed from the condvar (reset to `undef`), so the condvar does not keep a reference to the callback after invocation.

SUPPORTED EVENT LOOPS/BACKENDS

The following backend classes are part of the AnyEvent distribution (every class has its own manpage):

Backends that are autoprobed when no other event loop can be found.
EV is the preferred backend when no other event loop seems to be in use. If EV is not installed, then AnyEvent will fall back to its own pure-perl implementation, which is available everywhere as it comes with AnyEvent itself.

```
AnyEvent::Impl::EV        based on EV (interface to libev, best choice).
AnyEvent::Impl::Perl      pure-perl AnyEvent::Loop, fast and portable.
```

Backends that are transparently being picked up when they are used.
These will be used if they are already loaded when the first watcher is created, in which case it is assumed that the application is using them. This means that AnyEvent will automatically pick the right backend when the main program loads an event module before anything starts to create watchers. Nothing special needs to be done by the main program.

```
AnyEvent::Impl::Event     based on Event, very stable, few glitches.
AnyEvent::Impl::Glib      based on Glib, slow but very stable.
AnyEvent::Impl::Tk        based on Tk, very broken.
AnyEvent::Impl::UV        based on UV, innovated square wheels.
AnyEvent::Impl::EventLib  based on Event::Lib, leaks memory and worse.
AnyEvent::Impl::POE       based on POE, very slow, some limitations.
AnyEvent::Impl::Irssi     used when running within irssi.
AnyEvent::Impl::IOAsync   based on IO::Async.
AnyEvent::Impl::Cocoa     based on Cocoa::EventLoop.
AnyEvent::Impl::FLTK      based on FLTK (fltk 2 binding).
```

Backends with special needs.
Qt requires the Qt::Application to be instantiated first, but will otherwise be picked up automatically. As long as the main program instantiates the application before any AnyEvent watchers are created, everything should just work.

```
AnyEvent::Impl::Qt        based on Qt.
```

Event loops that are indirectly supported via other backends.
Some event loops can be supported via other modules:

There is no direct support for WxWidgets (Wx) or Prima.

WxWidgets has no support for watching file handles. However, you can use WxWidgets through the POE adaptor, as POE has a Wx backend that simply polls 20 times per second, which was considered to be too horrible to even consider for AnyEvent.

Prima is not supported as nobody seems to be using it, but it has a POE backend, so it can be supported through POE.

AnyEvent knows about both Prima and Wx, however, and will try to load POE when detecting them, in the hope that POE will pick them up, in which case everything will be automatic.

Known event loops outside the AnyEvent distribution
The following event loops or programs support AnyEvent by providing their own AnyEvent backend. They will be picked up automatically.

```
urxvt::anyevent           available to rxvt-unicode extensions
```

GLOBAL VARIABLES AND FUNCTIONS

These are not normally required to use AnyEvent, but can be useful to write AnyEvent extension modules.

`$AnyEvent::MODEL`
: Contains `undef` until the first watcher is being created, before the backend has been autodetected.

  Afterwards it contains the event model that is being used, which is the name of the Perl class implementing the model. This class is usually one of the `AnyEvent::Impl::xxx` modules, but can be any other class in the case AnyEvent has been extended at runtime (e.g. in *rxvt-unicode* it will be `urxvt::anyevent`).

AnyEvent::detect
: Returns `$AnyEvent::MODEL`, forcing autodetection of the event model if necessary. You should only call this function right before you would have created an AnyEvent watcher anyway, that is, as late as possible at runtime, and not e.g. during initialisation of your module.

  The effect of calling this function is as if a watcher had been created (specifically, actions that happen "when the first watcher is created" happen when calling detetc as well).

  If you need to do some initialisation before AnyEvent watchers are created, use `post_detect`.

`$guard` = AnyEvent::post_detect { BLOCK }
: Arranges for the code block to be executed as soon as the event model is autodetected (or immediately if that has already happened).

  The block will be executed *after* the actual backend has been detected (`$AnyEvent::MODEL` is set), so it is possible to do some initialisation only when AnyEvent is actually initialised – see the sources of AnyEvent::AIO to see how this is used.

  The most common usage is to create some global watchers, without forcing event module detection too early. For example, AnyEvent::AIO creates and installs the global IO::AIO watcher in a `post_detect` block to avoid autodetecting the event module at load time.

  If called in scalar or list context, then it creates and returns an object that automatically removes the callback again when it is destroyed (or `undef` when the hook was immediately executed). See AnyEvent::AIO for a case where this is useful.

  Example: Create a watcher for the IO::AIO module and store it in `$WATCHER`, but do so only do so after the event loop is initialised.

```
our WATCHER;

my $guard = AnyEvent::post_detect {
   $WATCHER = AnyEvent->io (fh => IO::AIO::poll_fileno, poll => 'r', cb =>
};

# the ||= is important in case post_detect immediately runs the block,
# as to not clobber the newly-created watcher. assigning both watcher and
# post_detect guard to the same variable has the advantage of users being
# able to just C<undef $WATCHER> if the watcher causes them grief.

$WATCHER ||= $guard;
```

`@AnyEvent::post_detect`
: This is a lower level interface then `AnyEvent::post_detect` (the function). This variable is mainly useful for modules that can do something useful when AnyEvent is used and thus want to know when it is initialised, but do not need to even load it by default. This array provides the means to hook into AnyEvent passively, without loading it.

  Here is how it works: If there are any code references in this array (you can `push` to it before or after loading AnyEvent), then they will be called directly after the event loop has been chosen.

You should check `$AnyEvent::MODEL` before adding to this array, though: if it is defined then the event loop has already been detected, and the array will be ignored.

Best use `AnyEvent::post_detect { BLOCK }` when your application allows it, as it takes care of these details.

Example: To load Coro::AnyEvent whenever Coro and AnyEvent are used together, you could put this into Coro (this is the actual code used by Coro to accomplish this):

```
if (defined $AnyEvent::MODEL) {
   # AnyEvent already initialised, so load Coro::AnyEvent
   require Coro::AnyEvent;
} else {
   # AnyEvent not yet initialised, so make sure to load Coro::AnyEvent
   # as soon as it is
   push @AnyEvent::post_detect, sub { require Coro::AnyEvent };
}
```

AnyEvent::postpone { BLOCK }

Arranges for the block to be executed as soon as possible, but not before the call itself returns. In practise, the block will be executed just before the event loop polls for new events, or shortly afterwards.

This function never returns anything (to make the `return postpone { ... }` idiom more useful.

To understand the usefulness of this function, consider a function that asynchronously does something for you and returns some transaction object or guard to let you cancel the operation. For example, `AnyEvent::Socket::tcp_connect`:

```
# start a connection attempt unless one is active
$self->{connect_guard} ||= AnyEvent::Socket::tcp_connect "www.example.net",
   delete $self->{connect_guard};
   ...
};
```

Imagine that this function could instantly call the callback, for example, because it detects an obvious error such as a negative port number. Invoking the callback before the function returns causes problems however: the callback will be called and will try to delete the guard object. But since the function hasn't returned yet, there is nothing to delete. When the function eventually returns it will assign the guard object to `$self->{connect_guard}`, where it will likely never be deleted, so the program thinks it is still trying to connect.

This is where `AnyEvent::postpone` should be used. Instead of calling the callback directly on error:

```
$cb->(undef), return # signal error to callback, BAD!
   if $some_error_condition;
```

It should use `postpone`:

```
AnyEvent::postpone { $cb->(undef) }, return # signal error to callback, lat
   if $some_error_condition;
```

AnyEvent::log `$level`, `$msg`[, `@args`]

Log the given `$msg` at the given `$level`.

If AnyEvent::Log is not loaded then this function makes a simple test to see whether the message will be logged. If the test succeeds it will load AnyEvent::Log and call `AnyEvent::Log::log` – consequently, look at the AnyEvent::Log documentation for details.

If the test fails it will simply return. Right now this happens when a numerical loglevel is used and it is larger than the level specified via `$ENV{PERL_ANYEVENT_VERBOSE}`.

If you want to sprinkle loads of logging calls around your code, consider creating a logger callback with the `AnyEvent::Log::logger` function, which can reduce typing, codesize and can reduce the logging overhead enormously.

AnyEvent::fh_block `$filehandle`
AnyEvent::fh_unblock `$filehandle`
Sets blocking or non-blocking behaviour for the given filehandle.

WHAT TO DO IN A MODULE

As a module author, you should `use AnyEvent` and call AnyEvent methods freely, but you should not load a specific event module or rely on it.

Be careful when you create watchers in the module body − AnyEvent will decide which event module to use as soon as the first method is called, so by calling AnyEvent in your module body you force the user of your module to load the event module first.

Never call `->recv` on a condition variable unless you *know* that the `->send` method has been called on it already. This is because it will stall the whole program, and the whole point of using events is to stay interactive.

It is fine, however, to call `->recv` when the user of your module requests it (i.e. if you create a http request object ad have a method called `results` that returns the results, it may call `->recv` freely, as the user of your module knows what she is doing. Always).

WHAT TO DO IN THE MAIN PROGRAM

There will always be a single main program − the only place that should dictate which event model to use.

If the program is not event-based, it need not do anything special, even when it depends on a module that uses an AnyEvent. If the program itself uses AnyEvent, but does not care which event loop is used, all it needs to do is `use AnyEvent`. In either case, AnyEvent will choose the best available loop implementation.

If the main program relies on a specific event model − for example, in Gtk2 programs you have to rely on the Glib module − you should load the event module before loading AnyEvent or any module that uses it: generally speaking, you should load it as early as possible. The reason is that modules might create watchers when they are loaded, and AnyEvent will decide on the event model to use as soon as it creates watchers, and it might choose the wrong one unless you load the correct one yourself.

You can chose to use a pure-perl implementation by loading the `AnyEvent::Loop` module, which gives you similar behaviour everywhere, but letting AnyEvent chose the model is generally better.

MAINLOOP EMULATION

Sometimes (often for short test scripts, or even standalone programs who only want to use AnyEvent), you do not want to run a specific event loop.

In that case, you can use a condition variable like this:

```
   AnyEvent->condvar->recv;
```

This has the effect of entering the event loop and looping forever.

Note that usually your program has some exit condition, in which case it is better to use the ''traditional'' approach of storing a condition variable somewhere, waiting for it, and sending it when the program should exit cleanly.

OTHER MODULES

The following is a non-exhaustive list of additional modules that use AnyEvent as a client and can therefore be mixed easily with other AnyEvent modules and other event loops in the same program. Some of the modules come as part of AnyEvent, the others are available via CPAN (see <http://search.cpan.org/search?m=module&q=anyevent%3A%3A*> for a longer non-exhaustive list), and the list is heavily biased towards modules of the AnyEvent author himself :)

AnyEvent::Util (part of the AnyEvent distribution)
: Contains various utility functions that replace often-used blocking functions such as `inet_aton` with event/callback–based versions.

AnyEvent::Socket (part of the AnyEvent distribution)
: Provides various utility functions for (internet protocol) sockets, addresses and name resolution. Also functions to create non-blocking tcp connections or tcp servers, with IPv6 and SRV record support and more.

AnyEvent::Handle (part of the AnyEvent distribution)
: Provide read and write buffers, manages watchers for reads and writes, supports raw and formatted I/O, I/O queued and fully transparent and non-blocking SSL/TLS (via AnyEvent::TLS).

AnyEvent::DNS (part of the AnyEvent distribution)
: Provides rich asynchronous DNS resolver capabilities.

AnyEvent::HTTP, AnyEvent::IRC, AnyEvent::XMPP, AnyEvent::GPSD, AnyEvent::IGS, AnyEvent::FCP
: Implement event-based interfaces to the protocols of the same name (for the curious, IGS is the International Go Server and FCP is the Freenet Client Protocol).

AnyEvent::AIO (part of the AnyEvent distribution)
: Truly asynchronous (as opposed to non-blocking) I/O, should be in the toolbox of every event programmer. AnyEvent::AIO transparently fuses IO::AIO and AnyEvent together, giving AnyEvent access to event-based file I/O, and much more.

AnyEvent::Fork, AnyEvent::Fork::RPC, AnyEvent::Fork::Pool, AnyEvent::Fork::Remote
: These let you safely fork new subprocesses, either locally or remotely (e.g.v ia ssh), using some RPC protocol or not, without the limitations normally imposed by fork (AnyEvent works fine for example). Dynamically-resized worker pools are obviously included as well.

  And they are quite tiny and fast as well – "abusing" AnyEvent::Fork just to exec external programs can easily beat using `fork` and `exec` (or even `system`) in most programs.

AnyEvent::Filesys::Notify
: AnyEvent is good for non-blocking stuff, but it can't detect file or path changes (e.g. "watch this directory for new files", "watch this file for changes"). The AnyEvent::Filesys::Notify module promises to do just that in a portbale fashion, supporting inotify on GNU/Linux and some weird, without doubt broken, stuff on OS X to monitor files. It can fall back to blocking scans at regular intervals transparently on other platforms, so it's about as portable as it gets.

  (I haven't used it myself, but it seems the biggest problem with it is it quite bad performance).

AnyEvent::DBI
: Executes DBI requests asynchronously in a proxy process for you, notifying you in an event-based way when the operation is finished.

AnyEvent::FastPing
: The fastest ping in the west.

Coro
: Has special support for AnyEvent via Coro::AnyEvent, which allows you to simply invert the flow control – don't call us, we will call you:

```
    async {
       Coro::AnyEvent::sleep 5; # creates a 5s timer and waits for it
       print "5 seconds later!\n";

       Coro::AnyEvent::readable *STDIN; # uses an I/O watcher
       my $line = <STDIN>; # works for ttys

       AnyEvent::HTTP::http_get "url", Coro::rouse_cb;
       my ($body, $hdr) = Coro::rouse_wait;
```

```
        };
```

SIMPLIFIED AE API

Starting with version 5.0, AnyEvent officially supports a second, much simpler, API that is designed to reduce the calling, typing and memory overhead by using function call syntax and a fixed number of parameters.

See the AE manpage for details.

ERROR AND EXCEPTION HANDLING

In general, AnyEvent does not do any error handling – it relies on the caller to do that if required. The AnyEvent::Strict module (see also the `PERL_ANYEVENT_STRICT` environment variable, below) provides strict checking of all AnyEvent methods, however, which is highly useful during development.

As for exception handling (i.e. runtime errors and exceptions thrown while executing a callback), this is not only highly event-loop specific, but also not in any way wrapped by this module, as this is the job of the main program.

The pure perl event loop simply re-throws the exception (usually within `condvar->recv`), the Event and EV modules call `$Event/EV::DIED->()`, Glib uses `install_exception_handler` and so on.

ENVIRONMENT VARIABLES

AnyEvent supports a number of environment variables that tune the runtime behaviour. They are usually evaluated when AnyEvent is loaded, initialised, or a submodule that uses them is loaded. Many of them also cause AnyEvent to load additional modules – for example, `PERL_ANYEVENT_DEBUG_WRAP` causes the AnyEvent::Debug module to be loaded.

All the environment variables documented here start with `PERL_ANYEVENT_`, which is what AnyEvent considers its own namespace. Other modules are encouraged (but by no means required) to use `PERL_ANYEVENT_SUBMODULE` if they have registered the AnyEvent::Submodule namespace on CPAN, for any submodule. For example, AnyEvent::HTTP could be expected to use `PERL_ANYEVENT_HTTP_PROXY` (it should not access env variables starting with `AE_`, see below).

All variables can also be set via the `AE_` prefix, that is, instead of setting `PERL_ANYEVENT_VERBOSE` you can also set `AE_VERBOSE`. In case there is a clash btween anyevent and another program that uses `AE_something` you can set the corresponding `PERL_ANYEVENT_something` variable to the empty string, as those variables take precedence.

When AnyEvent is first loaded, it copies all `AE_xxx` env variables to their `PERL_ANYEVENT_xxx` counterpart unless that variable already exists. If taint mode is on, then AnyEvent will remove *all* environment variables starting with `PERL_ANYEVENT_` from `%ENV` (or replace them with `undef` or the empty string, if the corresaponding `AE_` variable is set).

The exact algorithm is currently:

```
   1. if taint mode enabled, delete all PERL_ANYEVENT_xyz variables from %ENV
   2. copy over AE_xyz to PERL_ANYEVENT_xyz unless the latter already exists
   3. if taint mode enabled, set all PERL_ANYEVENT_xyz variables to undef.
```

This ensures that child processes will not see the `AE_` variables.

The following environment variables are currently known to AnyEvent:

`PERL_ANYEVENT_VERBOSE`

By default, AnyEvent will log messages with loglevel `4` (`error`) or higher (see AnyEvent::Log). You can set this environment variable to a numerical loglevel to make AnyEvent more (or less) talkative.

If you want to do more than just set the global logging level you should have a look at `PERL_ANYEVENT_LOG`, which allows much more complex specifications.

When set to `0` (`off`), then no messages whatsoever will be logged with everything else at defaults.

When set to `5` or higher (`warn`), AnyEvent warns about unexpected conditions, such as not being able to load the event model specified by `PERL_ANYEVENT_MODEL`, or a guard callback throwing an exception – this is the minimum recommended level for use during development.

When set to `7` or higher (info), AnyEvent reports which event model it chooses.

When set to `8` or higher (debug), then AnyEvent will report extra information on which optional modules it loads and how it implements certain features.

`PERL_ANYEVENT_LOG`

Accepts rather complex logging specifications. For example, you could log all `debug` messages of some module to stderr, warnings and above to stderr, and errors and above to syslog, with:

```
   PERL_ANYEVENT_LOG=Some::Module=debug,+log:filter=warn,+%syslog:%syslog=erro
```

For the rather extensive details, see AnyEvent::Log.

This variable is evaluated when AnyEvent (or AnyEvent::Log) is loaded, so will take effect even before AnyEvent has initialised itself.

Note that specifying this environment variable causes the AnyEvent::Log module to be loaded, while `PERL_ANYEVENT_VERBOSE` does not, so only using the latter saves a few hundred kB of memory unless a module explicitly needs the extra features of AnyEvent::Log.

`PERL_ANYEVENT_STRICT`

AnyEvent does not do much argument checking by default, as thorough argument checking is very costly. Setting this variable to a true value will cause AnyEvent to load `AnyEvent::Strict` and then to thoroughly check the arguments passed to most method calls. If it finds any problems, it will croak.

In other words, enables "strict" mode.

Unlike `use strict` (or its modern cousin, `use common::sense`, it is definitely recommended to keep it off in production. Keeping `PERL_ANYEVENT_STRICT=1` in your environment while developing programs can be very useful, however.

`PERL_ANYEVENT_DEBUG_SHELL`

If this env variable is nonempty, then its contents will be interpreted by `AnyEvent::Socket::parse_hostport` and `AnyEvent::Debug::shell` (after replacing every occurrence of `$$` by the process pid). The shell object is saved in `$AnyEvent::Debug::SHELL`.

This happens when the first watcher is created.

For example, to bind a debug shell on a unix domain socket in */tmp/debug<pid>.sock*, you could use this:

```
   PERL_ANYEVENT_DEBUG_SHELL=/tmp/debug\$\$.sock perlprog
   # connect with e.g.: socat readline /tmp/debug123.sock
```

Or to bind to tcp port 4545 on localhost:

```
   PERL_ANYEVENT_DEBUG_SHELL=127.0.0.1:4545 perlprog
   # connect with e.g.: telnet localhost 4545
```

Note that creating sockets in */tmp* or on localhost is very unsafe on multiuser systems.

`PERL_ANYEVENT_DEBUG_WRAP`

Can be set to `0`, `1` or `2` and enables wrapping of all watchers for debugging purposes. See `AnyEvent::Debug::wrap` for details.

`PERL_ANYEVENT_MODEL`

This can be used to specify the event model to be used by AnyEvent, before auto detection and –probing kicks in.

It normally is a string consisting entirely of ASCII letters (e.g. `EV` or `IOAsync`). The string `AnyEvent::Impl::` gets prepended and the resulting module name is loaded and – if the load was successful − used as event model backend. If it fails to load then AnyEvent will proceed with auto detection and −probing.

If the string ends with `::` instead (e.g. `AnyEvent::Impl::EV::`) then nothing gets prepended and the module name is used as-is (hint: `::` at the end of a string designates a module name and quotes it appropriately).

For example, to force the pure perl model (AnyEvent::Loop::Perl) you could start your program like this:

```
PERL_ANYEVENT_MODEL=Perl perl ...
```

`PERL_ANYEVENT_IO_MODEL`

The current file I/O model − see AnyEvent::IO for more info.

At the moment, only `Perl` (small, pure-perl, synchronous) and `IOAIO` (truly asynchronous) are supported. The default is `IOAIO` if AnyEvent::AIO can be loaded, otherwise it is `Perl`.

`PERL_ANYEVENT_PROTOCOLS`

Used by both AnyEvent::DNS and AnyEvent::Socket to determine preferences for IPv4 or IPv6. The default is unspecified (and might change, or be the result of auto probing).

Must be set to a comma-separated list of protocols or address families, current supported: `ipv4` and `ipv6`. Only protocols mentioned will be used, and preference will be given to protocols mentioned earlier in the list.

This variable can effectively be used for denial-of-service attacks against local programs (e.g. when setuid), although the impact is likely small, as the program has to handle connection and other failures anyways.

Examples: `PERL_ANYEVENT_PROTOCOLS=ipv4,ipv6` − prefer IPv4 over IPv6, but support both and try to use both. `PERL_ANYEVENT_PROTOCOLS=ipv4` − only support IPv4, never try to resolve or contact IPv6 addresses. `PERL_ANYEVENT_PROTOCOLS=ipv6,ipv4` support either IPv4 or IPv6, but prefer IPv6 over IPv4.

`PERL_ANYEVENT_HOSTS`

This variable, if specified, overrides the */etc/hosts* file used by AnyEvent::Socket`::resolve_sockaddr`, i.e. hosts aliases will be read from that file instead.

`PERL_ANYEVENT_EDNS0`

Used by AnyEvent::DNS to decide whether to use the EDNS0 extension for DNS. This extension is generally useful to reduce DNS traffic, especially when DNSSEC is involved, but some (broken) firewalls drop such DNS packets, which is why it is off by default.

Setting this variable to `1` will cause AnyEvent::DNS to announce EDNS0 in its DNS requests.

`PERL_ANYEVENT_MAX_FORKS`

The maximum number of child processes that `AnyEvent::Util::fork_call` will create in parallel.

`PERL_ANYEVENT_MAX_OUTSTANDING_DNS`

The default value for the `max_outstanding` parameter for the default DNS resolver − this is the maximum number of parallel DNS requests that are sent to the DNS server.

`PERL_ANYEVENT_MAX_SIGNAL_LATENCY`

Perl has inherently racy signal handling (you can basically choose between losing signals and memory corruption) − pure perl event loops (including `AnyEvent::Loop`, when `Async::Interrupt` isn't available) therefore have to poll regularly to avoid losing signals.

Some event loops are racy, but don't poll regularly, and some event loops are written in C but are still racy. For those event loops, AnyEvent installs a timer that regularly wakes up the event loop.

By default, the interval for this timer is `10` seconds, but you can override this delay with this environment variable (or by setting the `$AnyEvent::MAX_SIGNAL_LATENCY` variable before creating signal watchers).

Lower values increase CPU (and energy) usage, higher values can introduce long delays when reaping children or waiting for signals.

The AnyEvent::Async module, if available, will be used to avoid this polling (with most event loops).

`PERL_ANYEVENT_RESOLV_CONF`
: The absolute path to a *resolv.conf*–style file to use instead of */etc/resolv.conf* (or the OS-specific configuration) in the default resolver, or the empty string to select the default configuration.

`PERL_ANYEVENT_CA_FILE`, `PERL_ANYEVENT_CA_PATH`.
: When neither `ca_file` nor `ca_path` was specified during AnyEvent::TLS context creation, and either of these environment variables are nonempty, they will be used to specify CA certificate locations instead of a system-dependent default.

`PERL_ANYEVENT_AVOID_GUARD` and `PERL_ANYEVENT_AVOID_ASYNC_INTERRUPT`
: When these are set to `1`, then the respective modules are not loaded. Mostly good for testing AnyEvent itself.

SUPPLYING YOUR OWN EVENT MODEL INTERFACE

This is an advanced topic that you do not normally need to use AnyEvent in a module. This section is only of use to event loop authors who want to provide AnyEvent compatibility.

If you need to support another event library which isn't directly supported by AnyEvent, you can supply your own interface to it by pushing, before the first watcher gets created, the package name of the event module and the package name of the interface to use onto `@AnyEvent::REGISTRY`. You can do that before and even without loading AnyEvent, so it is reasonably cheap.

Example:

```
push @AnyEvent::REGISTRY, [urxvt => urxvt::anyevent::];
```

This tells AnyEvent to (literally) use the `urxvt::anyevent::` package/class when it finds the `urxvt` package/module is already loaded.

When AnyEvent is loaded and asked to find a suitable event model, it will first check for the presence of urxvt by trying to `use` the `urxvt::anyevent` module.

The class should provide implementations for all watcher types. See AnyEvent::Impl::EV (source code), AnyEvent::Impl::Glib (Source code) and so on for actual examples. Use `perldoc -m AnyEvent::Impl::Glib` to see the sources.

If you don't provide `signal` and `child` watchers than AnyEvent will provide suitable (hopefully) replacements.

The above example isn't fictitious, the *rxvt-unicode* (a.k.a. urxvt) terminal emulator uses the above line as-is. An interface isn't included in AnyEvent because it doesn't make sense outside the embedded interpreter inside *rxvt-unicode*, and it is updated and maintained as part of the *rxvt-unicode* distribution.

rxvt-unicode also cheats a bit by not providing blocking access to condition variables: code blocking while waiting for a condition will `die`. This still works with most modules/usages, and blocking calls must not be done in an interactive application, so it makes sense.

EXAMPLE PROGRAM

The following program uses an I/O watcher to read data from STDIN, a timer to display a message once per second, and a condition variable to quit the program when the user enters quit:

```
use AnyEvent;

my $cv = AnyEvent->condvar;

my $io_watcher = AnyEvent->io (
   fh   => \*STDIN,
   poll => 'r',
   cb   => sub {
      warn "io event <$_[0]>\n";   # will always output <r>
      chomp (my $input = <STDIN>); # read a line
      warn "read: $input\n";       # output what has been read
      $cv->send if $input =~ /^q/i; # quit program if /^q/i
   },
);

my $time_watcher = AnyEvent->timer (after => 1, interval => 1, cb => sub {
   warn "timeout\n"; # print 'timeout' at most every second
});

$cv->recv; # wait until user enters /^q/i
```

REAL-WORLD EXAMPLE

Consider the Net::FCP module. It features (among others) the following API calls, which are to freenet what HTTP GET requests are to http:

```
   my $data = $fcp->client_get ($url); # blocks

   my $transaction = $fcp->txn_client_get ($url); # does not block
   $transaction->cb ( sub { ... } ); # set optional result callback
   my $data = $transaction->result; # possibly blocks
```

The `client_get` method works like `LWP::Simple::get`: it requests the given URL and waits till the data has arrived. It is defined to be:

```
   sub client_get { $_[0]->txn_client_get ($_[1])->result }
```

And in fact is automatically generated. This is the blocking API of Net::FCP, and it works as simple as in any other, similar, module.

More complicated is `txn_client_get`: It only creates a transaction (completion, result, ...) object and initiates the transaction.

```
   my $txn = bless { }, Net::FCP::Txn::;
```

It also creates a condition variable that is used to signal the completion of the request:

```
   $txn->{finished} = AnyAvent->condvar;
```

It then creates a socket in non-blocking mode.

```
   socket $txn->{fh}, ...;
   fcntl $txn->{fh}, F_SETFL, O_NONBLOCK;
   connect $txn->{fh}, ...
      and !$!{EWOULDBLOCK}
      and !$!{EINPROGRESS}
      and Carp::croak "unable to connect: $!\n";
```

Then it creates a write-watcher which gets called whenever an error occurs or the connection succeeds:

```
   $txn->{w} = AnyEvent->io (fh => $txn->{fh}, poll => 'w', cb => sub { $txn->fh_
```

And returns this transaction object. The `fh_ready_w` callback gets called as soon as the event loop detects that the socket is ready for writing.

The `fh_ready_w` method makes the socket blocking again, writes the request data and replaces the watcher by a read watcher (waiting for reply data). The actual code is more complicated, but that doesn't matter for this example:

```
fcntl $txn->{fh}, F_SETFL, 0;
syswrite $txn->{fh}, $txn->{request}
   or die "connection or write error";
$txn->{w} = AnyEvent->io (fh => $txn->{fh}, poll => 'r', cb => sub { $txn->fh_
```

Again, `fh_ready_r` waits till all data has arrived, and then stores the result and signals any possible waiters that the request has finished:

```
sysread $txn->{fh}, $txn->{buf}, length $txn->{$buf};

if (end-of-file or data complete) {
  $txn->{result} = $txn->{buf};
  $txn->{finished}->send;
  $txb->{cb}->($txn) of $txn->{cb}; # also call callback
}
```

The `result` method, finally, just waits for the finished signal (if the request was already finished, it doesn't wait, of course, and returns the data:

```
$txn->{finished}->recv;
return $txn->{result};
```

The actual code goes further and collects all errors (`die`s, exceptions) that occurred during request processing. The `result` method detects whether an exception as thrown (it is stored inside the `$txn` object) and just throws the exception, which means connection errors and other problems get reported to the code that tries to use the result, not in a random callback.

All of this enables the following usage styles:

1. Blocking:

```
my $data = $fcp->client_get ($url);
```

2. Blocking, but running in parallel:

```
my @datas = map $_->result,
               map $fcp->txn_client_get ($_),
                  @urls;
```

Both blocking examples work without the module user having to know anything about events.

3a. Event-based in a main program, using any supported event module:

```
use EV;

$fcp->txn_client_get ($url)->cb (sub {
   my $txn = shift;
   my $data = $txn->result;
   ...
});

EV::run;
```

3b. The module user could use AnyEvent, too:

```
use AnyEvent;

my $quit = AnyEvent->condvar;

$fcp->txn_client_get ($url)->cb (sub {
   ...
   $quit->send;
});
```

```
   $quit->recv;
```

BENCHMARKS

To give you an idea of the performance and overheads that AnyEvent adds over the event loops themselves and to give you an impression of the speed of various event loops I prepared some benchmarks.

BENCHMARKING ANYEVENT OVERHEAD

Here is a benchmark of various supported event models used natively and through AnyEvent. The benchmark creates a lot of timers (with a zero timeout) and I/O watchers (watching STDOUT, a pty, to become writable, which it is), lets them fire exactly once and destroys them again.

Source code for this benchmark is found as *eg/bench* in the AnyEvent distribution. It uses the AE interface, which makes a real difference for the EV and Perl backends only.

Explanation of the columns

watcher is the number of event watchers created/destroyed. Since different event models feature vastly different performances, each event loop was given a number of watchers so that overall runtime is acceptable and similar between tested event loop (and keep them from crashing): Glib would probably take thousands of years if asked to process the same number of watchers as EV in this benchmark.

bytes is the number of bytes (as measured by the resident set size, RSS) consumed by each watcher. This method of measuring captures both C and Perl-based overheads.

create is the time, in microseconds (millionths of seconds), that it takes to create a single watcher. The callback is a closure shared between all watchers, to avoid adding memory overhead. That means closure creation and memory usage is not included in the figures.

invoke is the time, in microseconds, used to invoke a simple callback. The callback simply counts down a Perl variable and after it was invoked ‘‘watcher’’ times, it would `->send` a condvar once to signal the end of this phase.

destroy is the time, in microseconds, that it takes to destroy a single watcher.

Results

```
          name watchers bytes create invoke destroy comment
         EV/EV   100000   223   0.47   0.43    0.27 EV native interface
        EV/Any   100000   223   0.48   0.42    0.26 EV + AnyEvent watchers
  Coro::EV/Any   100000   223   0.47   0.42    0.26 coroutines + Coro::Signal
      Perl/Any   100000   431   2.70   0.74    0.92 pure perl implementation
   Event/Event    16000   516  31.16  31.84    0.82 Event native interface
     Event/Any    16000  1203  42.61  34.79    1.80 Event + AnyEvent watchers
   IOAsync/Any    16000  1911  41.92  27.45   16.81 via IO::Async::Loop::IO_Poll
   IOAsync/Any    16000  1726  40.69  26.37   15.25 via IO::Async::Loop::Epoll
      Glib/Any    16000  1118  89.00  12.57   51.17 quadratic behaviour
        Tk/Any     2000  1346  20.96  10.75    8.00 SEGV with >> 2000 watchers
       POE/Any     2000  6951 108.97 795.32   14.24 via POE::Loop::Event
       POE/Any     2000  6648  94.79 774.40  575.51 via POE::Loop::Select
```

Discussion

The benchmark does *not* measure scalability of the event loop very well. For example, a select-based event loop (such as the pure perl one) can never compete with an event loop that uses epoll when the number of file descriptors grows high. In this benchmark, all events become ready at the same time, so select/poll−based implementations get an unnatural speed boost.

Also, note that the number of watchers usually has a nonlinear effect on overall speed, that is, creating twice as many watchers doesn’t take twice the time − usually it takes longer. This puts event loops tested with a higher number of watchers at a disadvantage.

To put the range of results into perspective, consider that on the benchmark machine, handling an event takes roughly 1600 CPU cycles with EV, 3100 CPU cycles with AnyEvent's pure perl loop and almost 3000000 CPU cycles with POE.

`EV` is the sole leader regarding speed and memory use, which are both maximal/minimal, respectively. When using the AE API there is zero overhead (when going through the AnyEvent API create is about 5–6 times slower, with other times being equal, so still uses far less memory than any other event loop and is still faster than Event natively).

The pure perl implementation is hit in a few sweet spots (both the constant timeout and the use of a single fd hit optimisations in the perl interpreter and the backend itself). Nevertheless this shows that it adds very little overhead in itself. Like any select-based backend its performance becomes really bad with lots of file descriptors (and few of them active), of course, but this was not subject of this benchmark.

The `Event` module has a relatively high setup and callback invocation cost, but overall scores in on the third place.

`IO::Async` performs admirably well, about on par with `Event`, even when using its pure perl backend.

`Glib`'s memory usage is quite a bit higher, but it features a faster callback invocation and overall ends up in the same class as `Event`. However, Glib scales extremely badly, doubling the number of watchers increases the processing time by more than a factor of four, making it completely unusable when using larger numbers of watchers (note that only a single file descriptor was used in the benchmark, so inefficiencies of `poll` do not account for this).

The `Tk` adaptor works relatively well. The fact that it crashes with more than 2000 watchers is a big setback, however, as correctness takes precedence over speed. Nevertheless, its performance is surprising, as the file descriptor is **dup()**ed for each watcher. This shows that the **dup()** employed by some adaptors is not a big performance issue (it does incur a hidden memory cost inside the kernel which is not reflected in the figures above).

`POE`, regardless of underlying event loop (whether using its pure perl select-based backend or the Event module, the POE-EV backend couldn't be tested because it wasn't working) shows abysmal performance and memory usage with AnyEvent: Watchers use almost 30 times as much memory as EV watchers, and 10 times as much memory as Event (the high memory requirements are caused by requiring a session for each watcher). Watcher invocation speed is almost 900 times slower than with AnyEvent's pure perl implementation.

The design of the POE adaptor class in AnyEvent can not really account for the performance issues, though, as session creation overhead is small compared to execution of the state machine, which is coded pretty optimally within AnyEvent::Impl::POE (and while everybody agrees that using multiple sessions is not a good approach, especially regarding memory usage, even the author of POE could not come up with a faster design).

Summary

- Using EV through AnyEvent is faster than any other event loop (even when used without AnyEvent), but most event loops have acceptable performance with or without AnyEvent.
- The overhead AnyEvent adds is usually much smaller than the overhead of the actual event loop, only with extremely fast event loops such as EV does AnyEvent add significant overhead.
- You should avoid POE like the plague if you want performance or reasonable memory usage.

BENCHMARKING THE LARGE SERVER CASE

This benchmark actually benchmarks the event loop itself. It works by creating a number of "servers": each server consists of a socket pair, a timeout watcher that gets reset on activity (but never fires), and an I/O watcher waiting for input on one side of the socket. Each time the socket watcher reads a byte it will write that byte to a random other "server".

The effect is that there will be a lot of I/O watchers, only part of which are active at any one point (so there is a constant number of active fds for each loop iteration, but which fds these are is random). The timeout is reset each time something is read because that reflects how most timeouts work (and puts extra pressure on the event loops).

In this benchmark, we use 10000 socket pairs (20000 sockets), of which 100 (1%) are active. This mirrors the activity of large servers with many connections, most of which are idle at any one point in time.

Source code for this benchmark is found as *eg/bench2* in the AnyEvent distribution. It uses the AE interface, which makes a real difference for the EV and Perl backends only.

Explanation of the columns

sockets is the number of sockets, and twice the number of ‘‘servers’’ (as each server has a read and write socket end).

create is the time it takes to create a socket pair (which is nontrivial) and two watchers: an I/O watcher and a timeout watcher.

request, the most important value, is the time it takes to handle a single ‘‘request’’, that is, reading the token from the pipe and forwarding it to another server. This includes deleting the old timeout and creating a new one that moves the timeout into the future.

Results

```
     name sockets create  request
       EV   20000  62.66     7.99
     Perl   20000  68.32    32.64
  IOAsync   20000 174.06   101.15 epoll
  IOAsync   20000 174.67   610.84 poll
    Event   20000 202.69   242.91
     Glib   20000 557.01  1689.52
      POE   20000 341.54 12086.32 uses POE::Loop::Event
```

Discussion

This benchmark *does* measure scalability and overall performance of the particular event loop.

EV is again fastest. Since it is using epoll on my system, the setup time is relatively high, though.

Perl surprisingly comes second. It is much faster than the C−based event loops Event and Glib.

IO::Async performs very well when using its epoll backend, and still quite good compared to Glib when using its pure perl backend.

Event suffers from high setup time as well (look at its code and you will understand why). Callback invocation also has a high overhead compared to the `$_->() for ..`−style loop that the Perl event loop uses. Event uses select or poll in basically all documented configurations.

Glib is hit hard by its quadratic behaviour w.r.t. many watchers. It clearly fails to perform with many filehandles or in busy servers.

POE is still completely out of the picture, taking over 1000 times as long as EV, and over 100 times as long as the Perl implementation, even though it uses a C−based event loop in this case.

Summary

- The pure perl implementation performs extremely well.
- Avoid Glib or POE in large projects where performance matters.

BENCHMARKING SMALL SERVERS

While event loops should scale (and select-based ones do not...) even to large servers, most programs we (or I :) actually write have only a few I/O watchers.

In this benchmark, I use the same benchmark program as in the large server case, but it uses only eight ‘‘servers’’, of which three are active at any one time. This should reflect performance for a small server relatively well.

The columns are identical to the previous table.

Results

```
   name sockets create request
     EV      16  20.00    6.54
   Perl      16  25.75   12.62
  Event      16  81.27   35.86
   Glib      16  32.63   15.48
    POE      16 261.87  276.28 uses POE::Loop::Event
```

Discussion

The benchmark tries to test the performance of a typical small server. While knowing how various event loops perform is interesting, keep in mind that their overhead in this case is usually not as important, due to the small absolute number of watchers (that is, you need efficiency and speed most when you have lots of watchers, not when you only have a few of them).

EV is again fastest.

Perl again comes second. It is noticeably faster than the C−based event loops Event and Glib, although the difference is too small to really matter.

POE also performs much better in this case, but is is still far behind the others.

Summary

- C−based event loops perform very well with small number of watchers, as the management overhead dominates.

THE IO::Lambda BENCHMARK

Recently I was told about the benchmark in the IO::Lambda manpage, which could be misinterpreted to make AnyEvent look bad. In fact, the benchmark simply compares IO::Lambda with POE, and IO::Lambda looks better (which shouldn't come as a surprise to anybody). As such, the benchmark is fine, and mostly shows that the AnyEvent backend from IO::Lambda isn't very optimal. But how would AnyEvent compare when used without the extra baggage? To explore this, I wrote the equivalent benchmark for AnyEvent.

The benchmark itself creates an echo-server, and then, for 500 times, connects to the echo server, sends a line, waits for the reply, and then creates the next connection. This is a rather bad benchmark, as it doesn't test the efficiency of the framework or much non-blocking I/O, but it is a benchmark nevertheless.

```
name                    runtime
Lambda/select           0.330 sec
   + optimized          0.122 sec
Lambda/AnyEvent         0.327 sec
   + optimized          0.138 sec
Raw sockets/select      0.077 sec
POE/select, components  0.662 sec
POE/select, raw sockets 0.226 sec
POE/select, optimized   0.404 sec

AnyEvent/select/nb      0.085 sec
AnyEvent/EV/nb          0.068 sec
   +state machine       0.134 sec
```

The benchmark is also a bit unfair (my fault): the IO::Lambda/POE benchmarks actually make blocking connects and use 100% blocking I/O, defeating the purpose of an event-based solution. All of the newly written AnyEvent benchmarks use 100% non-blocking connects (using AnyEvent::Socket::tcp_connect and the asynchronous pure perl DNS resolver), so AnyEvent is at a disadvantage here, as non-blocking connects generally require a lot more bookkeeping and event handling than blocking connects (which involve a single syscall only).

The last AnyEvent benchmark additionally uses AnyEvent::Handle, which offers similar expressive power as POE and IO::Lambda, using conventional Perl syntax. This means that both the echo server and the client are 100% non-blocking, further placing it at a disadvantage.

As you can see, the AnyEvent + EV combination even beats the hand-optimised "raw sockets benchmark", while AnyEvent + its pure perl backend easily beats IO::Lambda and POE.

And even the 100% non-blocking version written using the high-level (and slow :) AnyEvent::Handle abstraction beats both POE and IO::Lambda higher level ("unoptimised") abstractions by a large margin, even though it does all of DNS, tcp-connect and socket I/O in a non-blocking way.

The two AnyEvent benchmarks programs can be found as *eg/ae0.pl* and *eg/ae2.pl* in the AnyEvent distribution, the remaining benchmarks are part of the IO::Lambda distribution and were used without any changes.

SIGNALS

AnyEvent currently installs handlers for these signals:

SIGCHLD

A handler for `SIGCHLD` is installed by AnyEvent's child watcher emulation for event loops that do not support them natively. Also, some event loops install a similar handler.

Additionally, when AnyEvent is loaded and SIGCHLD is set to IGNORE, then AnyEvent will reset it to default, to avoid losing child exit statuses.

SIGPIPE

A no-op handler is installed for `SIGPIPE` when `$SIG{PIPE}` is `undef` when AnyEvent gets loaded.

The rationale for this is that AnyEvent users usually do not really depend on SIGPIPE delivery (which is purely an optimisation for shell use, or badly-written programs), but `SIGPIPE` can cause spurious and rare program exits as a lot of people do not expect `SIGPIPE` when writing to some random socket.

The rationale for installing a no-op handler as opposed to ignoring it is that this way, the handler will be restored to defaults on exec.

Feel free to install your own handler, or reset it to defaults.

RECOMMENDED/OPTIONAL MODULES

One of AnyEvent's main goals is to be 100% Pure−Perl(tm): only perl (and its built-in modules) are required to use it.

That does not mean that AnyEvent won't take advantage of some additional modules if they are installed.

This section explains which additional modules will be used, and how they affect AnyEvent's operation.

Async::Interrupt

This slightly arcane module is used to implement fast signal handling: To my knowledge, there is no way to do completely race-free and quick signal handling in pure perl. To ensure that signals still get delivered, AnyEvent will start an interval timer to wake up perl (and catch the signals) with some delay (default is 10 seconds, look for `$AnyEvent::MAX_SIGNAL_LATENCY`).

If this module is available, then it will be used to implement signal catching, which means that signals will not be delayed, and the event loop will not be interrupted regularly, which is more efficient (and good for battery life on laptops).

This affects not just the pure-perl event loop, but also other event loops that have no signal handling on their own (e.g. Glib, Tk, Qt).

Some event loops (POE, Event, Event::Lib) offer signal watchers natively, and either employ their own workarounds (POE) or use AnyEvent's workaround (using `$AnyEvent::MAX_SIGNAL_LATENCY`). Installing Async::Interrupt does nothing for those backends.

EV This module isn't really "optional", as it is simply one of the backend event loops that AnyEvent can use. However, it is simply the best event loop available in terms of features, speed and stability: It supports the AnyEvent API optimally, implements all the watcher types in XS, does automatic timer adjustments even when no monotonic clock is available, can take avdantage of advanced kernel interfaces such as `epoll` and `kqueue`, and is the fastest backend *by far*. You can even embed Glib/Gtk2 in it (or vice versa, see EV::Glib and Glib::EV).

If you only use backends that rely on another event loop (e.g. `Tk`), then this module will do nothing for you.

Guard
: The guard module, when used, will be used to implement `AnyEvent::Util::guard`. This speeds up guards considerably (and uses a lot less memory), but otherwise doesn't affect guard operation much. It is purely used for performance.

JSON and JSON::XS
: One of these modules is required when you want to read or write JSON data via AnyEvent::Handle. JSON is also written in pure-perl, but can take advantage of the ultra-high-speed JSON::XS module when it is installed.

Net::SSLeay
: Implementing TLS/SSL in Perl is certainly interesting, but not very worthwhile: If this module is installed, then AnyEvent::Handle (with the help of AnyEvent::TLS), gains the ability to do TLS/SSL.

Time::HiRes
: This module is part of perl since release 5.008. It will be used when the chosen event library does not come with a timing source of its own. The pure-perl event loop (AnyEvent::Loop) will additionally load it to try to use a monotonic clock for timing stability.

AnyEvent::AIO (and IO::AIO)
: The default implementation of AnyEvent::IO is to do I/O synchronously, stopping programs while they access the disk, which is fine for a lot of programs.

Installing AnyEvent::AIO (and its IO::AIO dependency) makes it switch to a true asynchronous implementation, so event processing can continue even while waiting for disk I/O.

FORK

Most event libraries are not fork-safe. The ones who are usually are because they rely on inefficient but fork-safe `select` or `poll` calls − higher performance APIs such as BSD's kqueue or the dreaded Linux epoll are usually badly thought-out hacks that are incompatible with fork in one way or another. Only EV is fully fork-aware and ensures that you continue event-processing in both parent and child (or both, if you know what you are doing).

This means that, in general, you cannot fork and do event processing in the child if the event library was initialised before the fork (which usually happens when the first AnyEvent watcher is created, or the library is loaded).

If you have to fork, you must either do so *before* creating your first watcher OR you must not use AnyEvent at all in the child OR you must do something completely out of the scope of AnyEvent (see below).

The problem of doing event processing in the parent *and* the child is much more complicated: even for backends that *are* fork-aware or fork-safe, their behaviour is not usually what you want: fork clones all watchers, that means all timers, I/O watchers etc. are active in both parent and child, which is almost never what you want. Using `exec` to start worker children from some kind of manage prrocess is usually preferred, because it is much easier and cleaner, at the expense of having to have another binary.

In addition to logical problems with fork, there are also implementation problems. For example, on POSIX systems, you cannot fork at all in Perl code if a thread (I am talking of pthreads here) was ever created in the process, and this is just the tip of the iceberg. In general, using fork from Perl is difficult, and attempting to use fork without an exec to implement some kind of parallel processing is almost certainly doomed.

To safely fork and exec, you should use a module such as Proc::FastSpawn that lets you safely fork and exec new processes.

If you want to do multiprocessing using processes, you can look at the AnyEvent::Fork module (and some related modules such as AnyEvent::Fork::RPC, AnyEvent::Fork::Pool and AnyEvent::Fork::Remote). This module allows you to safely create subprocesses without any limitations − you can use X11 toolkits or AnyEvent in the children created by AnyEvent::Fork safely and without any special precautions.

SECURITY CONSIDERATIONS

AnyEvent can be forced to load any event model via `$ENV{PERL_ANYEVENT_MODEL}`. While this cannot (to my knowledge) be used to execute arbitrary code or directly gain access, it can easily be used to make the program hang or malfunction in subtle ways, as AnyEvent watchers will not be active when the program uses a different event model than specified in the variable.

You can make AnyEvent completely ignore this variable by deleting it before the first watcher gets created, e.g. with a `BEGIN` block:

```
BEGIN { delete $ENV{PERL_ANYEVENT_MODEL} }

use AnyEvent;
```

Similar considerations apply to `$ENV{PERL_ANYEVENT_VERBOSE}`, as that can be used to probe what backend is used and gain other information (which is probably even less useful to an attacker than PERL_ANYEVENT_MODEL), and `$ENV{PERL_ANYEVENT_STRICT}`.

Note that AnyEvent will remove *all* environment variables starting with `PERL_ANYEVENT_` from `%ENV` when it is loaded while taint mode is enabled.

BUGS

Perl 5.8 has numerous memleaks that sometimes hit this module and are hard to work around. If you suffer from memleaks, first upgrade to Perl 5.10 and check whether the leaks still show up. (Perl 5.10.0 has other annoying memleaks, such as leaking on `map` and `grep` but it is usually not as pronounced).

SEE ALSO

Tutorial/Introduction: AnyEvent::Intro.

FAQ: AnyEvent::FAQ.

Utility functions: AnyEvent::Util (misc. grab-bag), AnyEvent::Log (simply logging).

Development/Debugging: AnyEvent::Strict (stricter checking), AnyEvent::Debug (interactive shell, watcher tracing).

Supported event modules: AnyEvent::Loop, EV, EV::Glib, Glib::EV, Event, Glib::Event, Glib, Tk, Event::Lib, Qt, POE, FLTK, Cocoa::EventLoop, UV.

Implementations: AnyEvent::Impl::EV, AnyEvent::Impl::Event, AnyEvent::Impl::Glib, AnyEvent::Impl::Tk, AnyEvent::Impl::Perl, AnyEvent::Impl::EventLib, AnyEvent::Impl::Qt, AnyEvent::Impl::POE, AnyEvent::Impl::IOAsync, AnyEvent::Impl::Irssi, AnyEvent::Impl::FLTK, AnyEvent::Impl::Cocoa, AnyEvent::Impl::UV.

Non-blocking handles, pipes, stream sockets, TCP clients and servers: AnyEvent::Handle, AnyEvent::Socket, AnyEvent::TLS.

Asynchronous File I/O: AnyEvent::IO.

Asynchronous DNS: AnyEvent::DNS.

Thread support: Coro, Coro::AnyEvent, Coro::EV, Coro::Event.

Nontrivial usage examples: AnyEvent::GPSD, AnyEvent::IRC, AnyEvent::HTTP.

AUTHOR

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