NAME

AnyEvent::IO – the DBI of asynchronous I/O implementations

SYNOPSIS

```
use AnyEvent::IO;

# load /etc/passwd, call callback with the file data when done.
aio_load "/etc/passwd", sub {
   my ($data) = @_
      or return AE::log error => "/etc/passwd: $!";

   warn "/etc/passwd contains ", ($data =~ y/://) , " colons.\n";
};

# the rest of the SYNOPSIS does the same, but with individual I/O calls

# also import O_XXX flags
use AnyEvent::IO qw(:DEFAULT :flags);

my $filedata = AE::cv;

# first open the file
aio_open "/etc/passwd", O_RDONLY, 0, sub {
   my ($fh) = @_
      or return AE::log error => "/etc/passwd: $!";

   # now stat the file to get the size
   aio_stat $fh, sub {
      @_
         or return AE::log error => "/etc/passwd: $!";

      my $size = -s _;

      # now read all the file data
      aio_read $fh, $size, sub {
         my ($data) = @_
            or return AE::log error => "/etc/passwd: $!";

         $size == length $data
            or return AE::log error => "/etc/passwd: short read, file changed?

         # mostly the same as aio_load, above - $data contains
         # the file contents now.
         $filedata->($data);
      };
   };
};

my $passwd = $filedata->recv;
warn length $passwd, " octets.\n";
```

DESCRIPTION

This module provides functions that do I/O in an asynchronous fashion. It is to I/O the same as AnyEvent is to event libraries – it only *interfaces* to other implementations or to a portable pure-perl implementation (which does not, however, do asynchronous I/O).

The only other implementation that is supported (or even known to the author) is IO::AIO, which is used automatically when it can be loaded (via AnyEvent::AIO, which also needs to be installed). If it is not available, then AnyEvent::IO falls back to its synchronous pure-perl implementation.

Unlike AnyEvent, which model to use is currently decided at module load time, not at first use. Future releases might change this.

RATIONALE

While disk I/O often seems "instant" compared to, say, socket I/O, there are many situations where your program can block for extended time periods when doing disk I/O. For example, you access a disk on an NFS server and it is gone − can take ages to respond again, if ever. Or your system is extremely busy because it creates or restores a backup − reading data from disk can then take seconds. Or you use Linux, which for so many years has a close-to-broken VM/IO subsystem that can often induce minutes or more of delay for disk I/O, even under what I would consider light I/O loads.

Whatever the situation, some programs just can't afford to block for long times (say, half a second or more), because they need to respond as fast as possible.

For those cases, you need asynchronous I/O.

The problem is, AnyEvent itself sometimes reads disk files (for example, when looking at */etc/hosts*), and under the above situations, this can bring your program to a complete halt even if your program otherwise takes care to only use asynchronous I/O for everything (e.g. by using IO::AIO).

On the other hand, requiring IO::AIO for AnyEvent is clearly impossible, as AnyEvent promises to stay pure-perl, and the overhead of IO::AIO for small programs would be immense, especially when asynchronous I/O isn't even needed.

Clearly, this calls for an abstraction layer, and that is what you are looking at right now :−)

ASYNCHRONOUS VS. NON-BLOCKING

Many people are continuously confused on what the difference is between asynchronous I/O and non-blocking I/O. In fact, those two terms are not well defined, which often makes it hard to even talk about the difference. Here is a short guideline that should leave you less confused. It only talks about read operations, but the reasoning works with other I/O operations as well.

Non-blocking I/O means that data is delivered by some external means, automatically − that is, something *pushes* data towards your file handle, without you having to do anything. Non-blocking means that if your operating system currently has no data (or EOF, or some error) available for you, it will not wait ("block") as it would normally do, but immediately return with an error (e.g. `EWOULDBLOCK` − "I would have blocked, but you forbid it").

Your program can then wait for data to arrive by other means, for example, an I/O watcher which tells you when to re-attempt the read, after which it can try to read again, and so on.

Often, you would expect this to work for disk files as well − if the data isn't already in memory, one might want to wait for it and then re-attempt the read for example. While this is sound reasoning, the POSIX API does not support this, because disk drives and file systems do not send data "on their own", and more so, the OS already knows that data is there, it doesn't need to "wait" until it arrives from some external entity, it only needs to transfer the data from disk to your memory buffer.

So basically, while the concept is sound, the existing OS APIs do not support this. Therefore, it makes no sense to switch a disk file handle into non-blocking mode − it will behave exactly the same as in blocking mode, namely it will block until the data has been read from the disk.

The alternative to non-blocking I/O that actually works with disk files is usually called *asynchronous I/O*. Asynchronous, because the actual I/O is done while your program does something else: there is no need to call the read function to see if data is there, you only order the read once, and it will notify you when the read has finished and the data is your buffer − all the work is done in the background.

This works with disk files, and even with sockets and other sources. It is, however, not very efficient when used with sources that could be driven in a non-blocking way, because it usually has higher overhead in the OS than non-blocking I/O, because it ties memory buffers for a potentially unlimited time and often only a limited number of operations can be done in parallel.

That's why asynchronous I/O makes most sense when confronted with disk files, and non-blocking I/O only makes sense with sockets, pipes and similar streaming sources.

IMPORT TAGS

By default, this module exports all `aio_xxx` functions. In addition, the following import tags can be used:

```
:aio        all aio_* functions, same as :DEFAULT
:flags      the fcntl open flags (O_CREAT, O_RDONLY, ...)
```

API NOTES

The functions in this module are not meant to be the most versatile or the highest-performers (they are not very slow either, of course). They are primarily meant to give users of your code the option to do the I/O asynchronously (by installing IO::AIO and AnyEvent::AIO), without adding a dependency on those modules.

NAMING

All the functions in this module implement an I/O operation, usually with the same or similar name as the Perl built-in that they mimic, but with an `aio_` prefix. If you like you can think of the `aio_xxx` functions as "AnyEvent I/O" or "Asynchronous I/O" variants of Perl built-ins.

CALLING CONVENTIONS AND ERROR REPORTING

Each function expects a callback as their last argument. The callback is usually called with the result data or result code. An error is usually signalled by passing no arguments to the callback, which is then free to look at `$!` for the error code.

This makes all of the following forms of error checking valid:

```
aio_open ...., sub {
   my $fh = shift   # scalar assignment - will assign undef on error
      or return AE::log error => "...";

   my ($fh) = @_    # list assignment - will be 0 elements on error
      or return AE::log error => "...";

   @_               # check the number of elements directly
      or return AE::log error => "...";
```

CAVEAT: RELATIVE PATHS

When a path is specified, this path *must be an absolute* path, unless you make certain that nothing in your process calls `chdir` or an equivalent function while the request executes.

CAVEAT: OTHER SHARED STATE

Changing the `umask` while any requests execute that create files (or otherwise rely on the current umask) results in undefined behaviour − likewise changing anything else that would change the outcome, such as your effective user or group ID.

CALLBACKS MIGHT BE CALLED BEFORE FUNCTION RETURNS TO CALLER

Unlike other functions in the AnyEvent module family, these functions *may* call your callback instantly, before returning. This should not be a real problem, as these functions never return anything useful.

BEHAVIOUR AT PROGRAM EXIT

Both AnyEvent::IO::Perl and AnyEvent::IO::IOAIO implementations make sure that operations that have started will be finished on a clean programs exit. That makes programs work that start some I/O operations and then exit. For example this complete program:

```
use AnyEvent::IO;

aio_stat "path1", sub {
   aio_stat "path2", sub {
      warn "both stats done\n";
      };
   };
```

Starts a `stat` operation and then exits by "falling off the end" of the program. Nevertheless, *both* `stat` operations will be executed, as AnyEvent::IO waits for all outstanding requests to finish and you can start new requests from request callbacks.

In fact, since AnyEvent::IO::Perl is currently synchronous, the program will do both stats before falling off the end, but with AnyEvent::IO::IOAIO, the program first falls of the end, then the stats are executed.

While not guaranteed, this behaviour will be present in future versions, if reasonably possible (which is extreemly likely :).

GLOBAL VARIABLES AND FUNCTIONS

`$AnyEvent::IO::MODEL`

Contains the package name of the backend I/O model in use − at the moment, this is usually `AnyEvent::IO::Perl` or `AnyEvent::IO::IOAIO`.

aio_load `$path`, `$cb`−>($data)

Tries to open `$path` and read its contents into memory (obviously, should only be used on files that are "small enough"), then passes them to the callback as a string.

Example: load */etc/hosts*.

```
aio_load "/etc/hosts", sub {
   my ($hosts) = @_
      or return AE::log error => "/etc/hosts: $!";

   AE::log info => "/etc/hosts contains ", ($hosts =~ y/\n/), " lines\n";
};
```

aio_open `$path`, `$flags`, `$mode`, `$cb`−>($fh)

Tries to open the file specified by `$path` with the O_XXX−flags `$flags` (from the Fcntl module, or see below) and the mode `$mode` (a good value is 0666 for `O_CREAT`, and `0` otherwise).

The (normal, standard, perl) file handle associated with the opened file is then passed to the callback.

This works very much like Perl's `sysopen` function.

Changing the `umask` while this request executes results in undefined behaviour − likewise changing anything else that would change the outcome, such as your effective user or group ID.

To avoid having to load Fcntl, this module provides constants for `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_EXCL`, `O_TRUNC` and `O_APPEND` − you can either access them directly (`AnyEvent::IO::O_RDONLY`) or import them by specifying the `:flags` import tag (see SYNOPSIS).

Example: securely open a file in */var/tmp*, fail if it exists or is a symlink.

```
use AnyEvent::IO qw(:flags);

aio_open "/var/tmp/mytmp$$", O_CREAT | O_EXCL | O_RDWR, 0600, sub {
   my ($fh) = @_
      or return AE::log error => "$! - denial of service attack?";

   # now we have $fh
};
```

aio_close `$fh`, `$cb`−>($success)

Closes the file handle (yes, close can block your process indefinitely) and passes a true value to the callback on success.

Due to idiosyncrasies in perl, instead of calling `close`, the file handle might get closed by `dup2`'ing another file descriptor over it, that is, the `$fh` might still be open, but can be closed safely afterwards and must not be used for anything.

Example: close a file handle, and dirty as we are, do not even bother to check for errors.

```
aio_close $fh, sub { };
```

aio_read `$fh`, `$length`, `$cb`−>($data)

Tries to read `$length` octets from the current position from `$fh` and passes these bytes to `$cb`. Otherwise the semantics are very much like those of Perl's `sysread`.

If less than `$length` octets have been read, `$data` will contain only those bytes actually read. At EOF, `$data` will be a zero-length string. If an error occurs, then nothing is passed to the callback.

Obviously, multiple `aio_read`'s or `aio_write`'s at the same time on file handles sharing the underlying open file description results in undefined behaviour, due to sharing of the current file offset (and less obviously so, because OS X is not thread safe and corrupts data when you try).

Example: read 128 octets from a file.

```
aio_read $fh, 128, sub {
   my ($data) = @_
      or return AE::log error "read from fh: $!";

   if (length $data) {
      print "read ", length $data, " octets.\n";
   } else {
      print "EOF\n";
   }
};
```

aio_seek `$fh`, `$offset`, `$whence`, `$callback`−>($offs)

Seeks the filehandle to the new `$offset`, similarly to Perl's `sysseek`. The `$whence` are the traditional values (`0` to count from start, `1` to count from the current position and `2` to count from the end).

The resulting absolute offset will be passed to the callback on success.

Example: measure the size of the file in the old-fashioned way using seek.

```
aio_seek $fh, 0, 2, sub {
   my ($size) = @_
      or return AE::log error => "seek to end failed: $!";

   # maybe we need to seek to the beginning again?
   aio_seek $fh, 0, 0, sub {
      # now we are hopefully at the beginning
   };
};
```

aio_write `$fh`, `$data`, `$cb`−>($length)

Tries to write the octets in `$data` to the current position of `$fh` and passes the actual number of bytes written to the `$cb`. Otherwise the semantics are very much like those of Perl's `syswrite`.

If less than `length $data` octets have been written, `$length` will reflect that. If an error occurs, then nothing is passed to the callback.

Obviously, multiple `aio_read`'s or `aio_write`'s at the same time on file handles sharing the underlying open file description results in undefined behaviour, due to sharing of the current file offset (and less obviously so, because OS X is not thread safe and corrupts data when you try).

aio_truncate `$fh_or_path`, `$new_length`, `$cb`−>($success)
: Calls `truncate` on the path or perl file handle and passes a true value to the callback on success.

  Example: truncate */etc/passwd* to zero length − this only works on systems that support `truncate`, should not be tried out for obvious reasons and debian will probably open yte another security bug about this example.

```
aio_truncate "/etc/passwd", sub {
   @_
      or return AE::log error => "/etc/passwd: $! - are you root enough?";
};
```

aio_utime `$fh_or_path`, `$atime`, `$mtime`, `$cb`−>($success)
: Calls `utime` on the path or perl file handle and passes a true value to the callback on success.

  The special case of both `$atime` and `$mtime` being `undef` sets the times to the current time, on systems that support this.

  Example: try to touch *file*.

```
aio_utime "file", undef, undef, sub { };
```

aio_chown `$fh_or_path`, `$uid`, `$gid`, `$cb`−>($success)
: Calls `chown` on the path or perl file handle and passes a true value to the callback on success.

  If `$uid` or `$gid` can be specified as `undef`, in which case the uid or gid of the file is not changed. This differs from Perl's `chown` built-in, which wants `-1` for this.

  Example: update the group of *file* to 0 (root), but leave the owner alone.

```
aio_chown "file", undef, 0, sub {
   @_
      or return AE::log error => "chown 'file': $!";
};
```

aio_chmod `$fh_or_path`, `$perms`, `$cb`−>($success)
: Calls `chmod` on the path or perl file handle and passes a true value to the callback on success.

  Example: change *file* to be user/group/world−readable, but leave the other flags alone.

```
aio_stat "file", sub {
   @_
      or return AE::log error => "file: $!";

   aio_chmod "file", (stat _)[2] & 07777 | 00444, sub { };
};
```

aio_stat `$fh_or_path`, `$cb`−>($success)
aio_lstat `$path`, `$cb`−>($success)
: Calls `stat` or `lstat` on the path or perl file handle and passes a true value to the callback on success.

  The stat data will be available by `stat`'ing the _ file handle (e.g. `-x _`, `stat _` and so on).

  Example: see if we can find the number of subdirectories of */etc*.

```
aio_stat "/etc", sub {
   @_
      or return AE::log error => "/etc: $!";

   (stat _)[3] >= 2
      or return AE::log warn => "/etc has low link count - non-POSIX filesy
```

```
         print "/etc has ", (stat _)[3] - 2, " subdirectories.\n";
      };
```

aio_link `$oldpath`, `$newpath`, `$cb`->($success)
    Calls `link` on the paths and passes a true value to the callback on success.

    Example: link "*file* to *file.bak*, then rename *file.new* over *file*, to atomically replace it.

```
       aio_link "file", "file.bak", sub {
          @_
             or return AE::log error => "file: $!";

          aio_rename "file.new", "file", sub {
             @_
                or return AE::log error => "file.new: $!";

             print "file atomically replaced by file.new, backup file.bak\n";
          };
       };
```

aio_symlink `$oldpath`, `$newpath`, `$cb`->($success)
    Calls `symlink` on the paths and passes a true value to the callback on success.

    Example: create a symlink "*slink* containing "random data".

```
       aio_symlink "random data", "slink", sub {
          @_
             or return AE::log error => "slink: $!";
       };
```

aio_readlink `$path`, `$cb`->($target)
    Calls `readlink` on the paths and passes the link target string to the callback.

    Example: read the symlink called Fyslink> and verify that it contains "random data".

```
      aio_readlink "slink", sub {
         my ($target) = @_
            or return AE::log error => "slink: $!";

         $target eq "random data"
            or AE::log critical => "omg, the world will end!";
      };
```

aio_rename `$oldpath`, `$newpath`, `$cb`->($success)
    Calls `rename` on the paths and passes a true value to the callback on success.

    See `aio_link` for an example.

aio_unlink `$path`, `$cb`->($success)
    Tries to unlink the object at `$path` and passes a true value to the callback on success.

    Example: try to delete the file *tmpfile.dat~*.

```
       aio_unlink "tmpfile.dat~", sub { };
```

aio_mkdir `$path`, `$perms`, `$cb`->($success)
    Calls `mkdir` on the path with the given permissions `$perms` (when in doubt, `0777` is a good value) and passes a true value to the callback on success.

    Example: try to create the directory *subdir* and leave it to whoeveer comes after us to check whether it worked.

```
       aio_mkdir "subdir", 0777, sub { };
```

aio_rmdir `$path`, `$cb`−>($success)
: Tries to remove the directory at `$path` and passes a true value to the callback on success.

Example: try to remove the directory *subdir* and don't give a damn if that fails.

```
aio_rmdir "subdir", sub { };
```

aio_readdir `$path`, `$cb`−>(\@names)
: Reads all filenames from the directory specified by `$path` and passes them to the callback, as an array reference with the names (without a path prefix). The . and .. names will be filtered out first.

The ordering of the file names is undefined − backends that are capable of it (e.g. IO::AIO) will return the ordering that most likely is fastest to `stat` through, and furthermore put entries that likely are directories first in the array.

If you need best performance in recursive directory traversal or when looking at really big directories, you are advised to use IO::AIO directly, specifically the `aio_readdirx` and `aio_scandir` functions, which have more options to tune performance.

Example: recursively scan a directory hierarchy, silently skip diretcories we couldn't read and print all others.

```
sub scan($); # visibility-in-next statement is not so useful these days
sub scan($) {
   my ($path) = @_;

   aio_readdir $path, sub {
      my ($names) = @_
         or return;

      print "$path\n";

      for my $name (@$names) {
         aio_lstat "$path/$name", sub {
            scan "$path/$name"
               if -d _;
         };
      }
   };
}

scan "/etc";
```

ENVIRONMENT VARIABLES

See the description of `PERL_ANYEVENT_IO_MODEL` in the AnyEvent manpage.

AUTHOR

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