

NAME

"IO::Async::OS" – operating system abstractions for "IO::Async"

DESCRIPTION

This module acts as a class to provide a number of utility methods whose exact behaviour may depend on the type of OS it is running on. It is provided as a class so that specific kinds of operating system can override methods in it.

As well as these support functions it also provides a number of constants, all with names beginning `HAVE_` which describe various features that may or may not be available on the OS or perl build. Most of these are either hard-coded per OS, or detected at runtime.

The following constants may be overridden by environment variables.

- `HAVE_POSIX_FORK`

True if the `fork()` call has full POSIX semantics (full process separation). This is true on most OSes but false on MSWin32.

This may be overridden to be false by setting the environment variable `IO_ASYNC_NO_FORK`.

- `HAVE_THREADS`

True if `ithreads` are available, meaning that the `threads` module can be used. This depends on whether perl was built with threading support.

This may be overridable to be false by setting the environment variable `IO_ASYNC_NO_THREADS`.

getfamilybyname

```
$family = IO::Async::OS->getfamilybyname( $name )
```

Return a protocol family value based on the given name. If `$name` looks like a number it will be returned as-is. The string values `inet`, `inet6` and `unix` will be converted to the appropriate `AF_*` constant.

getsocktypebyname

```
$socktype = IO::Async::OS->getsocktypebyname( $name )
```

Return a socket type value based on the given name. If `$name` looks like a number it will be returned as-is. The string values `stream`, `dgram` and `raw` will be converted to the appropriate `SOCK_*` constant.

socketpair

```
( $S1, $S2 ) = IO::Async::OS->socketpair( $family, $socktype, $proto )
```

An abstraction of the `socketpair(2)` syscall, where any argument may be missing (or given as `undef`).

If `$family` is not provided, a suitable value will be provided by the OS (likely `AF_UNIX` on POSIX-based platforms). If `$socktype` is not provided, then `SOCK_STREAM` will be used.

Additionally, this method supports building connected `SOCK_STREAM` or `SOCK_DGRAM` pairs in the `AF_INET` family even if the underlying platform's `socketpair(2)` does not, by connecting two normal sockets together.

`$family` and `$socktype` may also be given symbolically as defined by `getfamilybyname` and `getsocktypebyname`.

pipepair

```
( $rd, $wr ) = IO::Async::OS->pipepair
```

An abstraction of the `pipe(2)` syscall, which returns the two new handles.

pipequad

```
( $rdA, $wrA, $rdB, $wrB ) = IO::Async::OS->pipequad
```

This method is intended for creating two pairs of filehandles that are linked together, suitable for passing as the `STDIN/STDOUT` pair to a child process. After this function returns, `$rdA` and `$wrA` will be a linked pair, as will `$rdB` and `$wrB`.

On platforms that support `socketpair(2)`, this implementation will be preferred, in which case `$rdA` and `$wrB` will actually be the same filehandle, as will `$rdB` and `$wrA`. This saves a file descriptor in the parent process.

When creating a `IO::Async::Stream` or subclass of it, the `read_handle` and `write_handle` parameters should always be used.

```
my ( $childRd, $myWr, $myRd, $childWr ) = IO::Async::OS->pipequad;

$loop->open_process(
    stdin => $childRd,
    stdout => $childWr,
    ...
);

my $str = IO::Async::Stream->new(
    read_handle => $myRd,
    write_handle => $myWr,
    ...
);
$loop->add( $str );
```

signame2num

```
$signum = IO::Async::OS->signame2num( $signame )
```

This utility method converts a signal name (such as “TERM”) into its system– specific signal number. This may be useful to pass to `POSIX::SigSet` or use in other places which use numbers instead of symbolic names.

extract_addrinfo

```
( $family, $socktype, $protocol, $addr ) = IO::Async::OS->extract_addrinfo( $ai )
```

Given an ARRAY or HASH reference value containing an `addrinfo`, returns a family, socktype and protocol argument suitable for a socket call and an address suitable for `connect` or `bind`.

If given an ARRAY it should be in the following form:

```
[ $family, $socktype, $protocol, $addr ]
```

If given a HASH it should contain the following keys:

```
family socktype protocol addr
```

Each field in the result will be initialised to 0 (or empty string for the address) if not defined in the `$ai` value.

The family type may also be given as a symbolic string as defined by `getfamilybyname`.

The socktype may also be given as a symbolic string; `stream`, `dgram` or `raw`; this will be converted to the appropriate `SOCK_*` constant.

Note that the `addr` field, if provided, must be a packed socket address, such as returned by `pack_sockaddr_in` or `pack_sockaddr_un`.

If the HASH form is used, rather than passing a packed socket address in the `addr` field, certain other hash keys may be used instead for convenience on certain named families.

```
family => 'inet'
```

Will pack an IP address and port number from keys called `ip` and `port`. If `ip` is missing it will be set to “0.0.0.0”. If `port` is missing it will be set to 0.

```
family => 'inet6'
```

Will pack an IP address and port number from keys called `ip` and `port`. If `ip` is missing it will be set to “:”. If `port` is missing it will be set to 0. Optionally will also include values from `scopeid` and `flowinfo` keys if provided.

This will only work if a `pack_sockaddr_in6` function can be found in `Socket`
family => 'unix'
Will pack a UNIX socket path from a key called `path`.

LOOP IMPLEMENTATION METHODS

The following methods are provided on `IO::Async::OS` because they are likely to require OS-specific implementations, but are used by `IO::Async::Loop` to implement its functionality. It can use the `HASH` reference `$loop->{os}` to store other data it requires.

loop_watch_signal

loop_unwatch_signal

```
IO::Async::OS->loop_watch_signal( $loop, $signal, $code )
```

```
IO::Async::OS->loop_unwatch_signal( $loop, $signal )
```

Used to implement the `watch_signal` / `unwatch_signal` `Loop` pair.

potentially_open_fds

```
@fds = IO::Async::OS->potentially_open_fds
```

Returns a list of filedescriptors which might need closing. By default this will return `0 .. _SC_OPEN_MAX`. OS-specific subclasses may have a better guess.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>