NAME

AnyEvent::Impl::POE − AnyEvent adaptor for POE

SYNOPSIS

```
use AnyEvent;
use POE;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make POE work with AnyEvent except by loading POE before creating the first AnyEvent watcher. There are some cases where POE will issue spurious (and non-suppressible) warnings. These can be avoided by loading AnyEvent::Impl::POE before loading any other modules using POE and AnyEvent, i.e. in your main program.

AnyEvent::Impl::POE will output some spurious message how to work around POE's spurious messages when it detects these cases.

Unfortunately, POE isn't generic enough to implement a fully working AnyEvent backend: POE is too badly designed, too badly documented and too badly implemented.

Here are the details, and what it means to you if you want to be interoperable with POE:

Weird messages

If you only use `run_one_timeslice` (as AnyEvent has to for its condition variables), POE will print an ugly, unsuppressible, message at program exit:

```
Sessions were started, but POE::Kernel's run() method was never...
```

The message is correct, the question is why POE prints it in the first place in a correct program (this is not a singular case though).

AnyEvent consequently patches the POE kernel so it thinks it already ran. Other workarounds, even the one cited in the POE documentation itself, have serious side effects, such as throwing away events.

The author of POE verified that this is indeed true, and has no plans to change this.

POE has other weird messages, and sometimes weird behaviour, for example, it doesn't support overloaded code references as callbacks for no apparent reason.

One POE session per Event

AnyEvent has to create one POE::Session per event watcher, which is immensely slow and makes watchers very large. The reason for this is lacking lifetime management (mostly undocumented, too). Without one session/watcher it is not possible to easily keep the kernel from running endlessly.

This is not just a problem with the way AnyEvent has to interact with POE, but is a principal issue with POEs lifetime management (namely that stopping the kernel stops sessions, but AnyEvent has no control over who and when the kernel starts or stops w.r.t. AnyEvent watcher creation/destruction).

From benchmark data it is not clear that session creation is that costly, though − the real inefficiencies with POE seem to come from other sources, such as event handling.

One watcher per fd/event combo

POE, of course, suffers from the same bug as Tk and some other badly designed event models in that it doesn't support multiple watchers per fd/poll combo. The workaround is the same as with Tk: AnyEvent::Impl::POE creates a separate file descriptor to hand to POE, which isn't fast and certainly not nice to your resources.

Of course, without the workaround, POE also prints ugly messages again that say the program *might* be buggy.

While this is not good to performance, at least regarding speed, with a modern Linux kernel, the overhead is actually quite small.

Timing deficiencies

POE manages to not have a function that returns the current time. This is extremely problematic, as POE can use different time functions, which can differ by more than a second – and user code is left guessing which one is used.

In addition, most timer functions in POE want an absolute timestamp, which is hard to create if all you have is a relative time and no function to return the ''current time''.

And of course POE doesn't handle time jumps at all (not even when using an event loop that happens to do that, such as EV, as it does its own unoptimised timer management).

AnyEvent works around the unavailability of the current time using relative timers exclusively, in the hope that POE gets it right at least internally.

Lack of defined event ordering

POE cannot guarantee the order of callback invocation for timers, and usually gets it wrong. That is, if you have two timers, one timing out after another (all else being equal), the callbacks might be called in reverse order.

How one manages to even implement stuff that way escapes me.

Child watchers

POE offers child watchers – which is a laudable thing, as few event loops do. Unfortunately, they cannot even implement AnyEvent's simple child watchers: they are not generic enough (the POE implementation isn't even generic enough to let properly designed back-end use their native child watcher instead – it insist on doing it itself the broken way).

Unfortunately, POE's child handling is inherently racy: if the child exits before the handler is created (because e.g. it crashes or simply is quick about it), then current versions of POE (1.352) will *never* invoke the child watcher, and there is nothing that can be done about it. Older versions of POE only delayed in this case. The reason is that POE first checks if the child has already exited, and *then* installs the signal handler – aa classical race.

Your only hope is for the fork'ed process to not exit too quickly, in which case everything happens to work.

Of course, whenever POE reaps an unrelated child it will also output a message for it that you cannot suppress (which shouldn't be too surprising at this point). Very professional.

As a workaround, AnyEvent::Impl::POE will take advantage of undocumented behaviour in POE::Kernel to catch the status of all child processes, but it cannot guarantee delivery.

How one manages to have such a glaring bug in an event loop after ten years of development escapes me.

(There are more annoying bugs, for example, POE runs `waitpid` unconditionally at finaliser time, so your program will hang until all child processes have exited.)

Documentation quality

At the time of this writing, POE was in its tenth year. Still, its documentation is extremely lacking, making it impossible to implement stuff as trivial as AnyEvent watchers without having to resort to undocumented behaviour or features.

For example, the POE::Kernel manpage has nine occurrences of the word TODO with an explanation of whats missing. In general, the POE man pages are littered with comments like ''section not yet written''.

Some other gems:

```
This allows many object methods to also be package methods.
```

This is nice, but since it doesn't document *which* methods these are, this is utterly useless information.

> Terminal signals will kill sessions if they are not handled by a "sig_handled"() call. The OS signals that usually kill or dump a process are considered terminal in POE, but they never trigger a coredump. These are: HUP, INT, QUIT and TERM.

Although AnyEvent calls `sig_handled`, removing it has no apparent effects on POE handling SIGINT.

> refcount_increment SESSION_ID, COUNTER_NAME

Nowhere is explained which COUNTER_NAMEs are valid and which aren't – not all scalars (or even strings) are valid counter names. Take your guess, failure is of course completely silent. I found this out the hard way, as the first name I came up with was silently ignored.

> get_next_event_time() returns the time the next event is due, in a form compatible with the UNIX time() function.

And surely, one would hope that POE supports sub-second accuracy as documented elsewhere, unlike the explanation above implies. Yet:

> POE::Kernel timers support subsecond accuracy, but don't expect too much here. Perl is not the right language for realtime programming.

... of course, Perl is not the right language to expect sub-second accuracy – the manpage author must hate Perl to spread so much FUD in so little space. The Deliantra game server logs with 100µs–accuracy because Perl is fast enough to require this, and is still able to deliver map updates with little jitter at exactly the right time. It does not, however, use POE.

> Furthermore, since the Kernel keeps track of everything sessions do, it knows when a session has run out of tasks to perform.

This is impossible – how does the kernel know that a session is no longer watching for some (external) event (e.g. by some other session)? It cannot, and therefore this is wrong – but you would be hard pressed to find out how to work around this and tell the kernel manually about such events.

It gets worse, though – the notion of ''task'' or ''resource'', although used throughout the documentation, is not defined in a usable way. For example, waiting for a timeout is considered to be a task, waiting for a signal is not (a session that only waits for a signal is considered finished and gets removed). The user is left guessing when waiting for an event counts as task and when not (in fact, the issue with signals is mentioned in passing in a section about child watchers and directly contradicts earlier parts in that document).

One could go on endlessly – ten years, no usable documentation.

It is likely that differences between documentation, or the one or two things I had to guess, cause unanticipated problems with this adaptor.

Fragile and inconsistent API

The POE API is extremely inconsistent – sometimes you have to pass a session argument, sometimes it gets ignored, sometimes a session-specific method must not use a session argument.

Error handling is sub-standard as well: even for programming mistakes, POE does not `croak` but, in most cases, just sets `$!` or simply does nothing at all, leading to fragile programs.

Sometimes registering a handler uses the ''eventname, parameter'' ordering (timeouts), sometimes it is ''parameter, eventname'' (signals). There is little consistency overall.

Lack of knowledge

> The IO::Poll event loop provides an alternative that theoretically scales better than select().

The IO::Poll ''event loop'' (who in his right mind would call that an event loop) of course scales about identically (sometimes it is a bit faster, sometimes a bit slower) to select in theory, and also in practise,

of course, as both are O(n) in the number of file descriptors, which is rather bad.

This is just one place where it gets obvious how little the author of the POE manpage understands.

No idle events
: The POE-recommended workaround to this is apparently to use `fork`. Consequently, idle watchers will have to be emulated by AnyEvent.

Questionable maintainer behaviour
: The author of POE is known to fabricate statements and post these to public mailinglists – apparently, spreading FUD about competing (in his eyes) projects or their maintainers is acceptable to him.

  This has (I believe) zero effects on the quality or usefulness of his code, but it does completely undermine his trustworthyness – so don't blindly believe anything he says, he might have just made it up to suit his needs (benchmark results, the names of my ten wifes, the length of my penis, etc. etc.). When in doubt, double-check – not just him, anybody actually.

  Example: <http://www.nntp.perl.org/group/perl.perl5.porters/2012/01/msg182141.html>. I challenged him in that thread to provide evidence for his statement by giving at least two examples, but of course since he just made it up, he couldn't provide any evidence.

On the good side, AnyEvent allows you to write your modules in a 100% POE-compatible way (bug-for-bug compatible even), without forcing your module to use POE – it is still open to better event models, of which there are plenty.

Oh, and one other positive thing:

```
RUNNING_IN_HELL
```

POE knows about the nature of the beast!

SEE ALSO

AnyEvent, POE.

AUTHOR

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