NAME

Async::Interrupt – allow C/XS libraries to interrupt perl asynchronously

SYNOPSIS

```
use Async::Interrupt;
```

DESCRIPTION

This module implements a single feature only of interest to advanced perl modules, namely asynchronous interruptions (think ''UNIX signals'', which are very similar).

Sometimes, modules wish to run code asynchronously (in another thread, or from a signal handler), and then signal the perl interpreter on certain events. One common way is to write some data to a pipe and use an event handling toolkit to watch for I/O events. Another way is to send a signal. Those methods are slow, and in the case of a pipe, also not asynchronous – it won't interrupt a running perl interpreter.

This module implements asynchronous notifications that enable you to signal running perl code from another thread, asynchronously, and sometimes even without using a single syscall.

USAGE SCENARIOS

Race-free signal handling

There seems to be no way to do race-free signal handling in perl: to catch a signal, you have to execute Perl code, and between entering the interpreter `select` function (or other blocking functions) and executing the select syscall is a small but relevant timespan during which signals will be queued, but perl signal handlers will not be executed and the blocking syscall will not be interrupted.

You can use this module to bind a signal to a callback while at the same time activating an event pipe that you can `select` on, fixing the race completely.

This can be used to implement the signal handling in event loops, e.g. AnyEvent, POE, IO::Async::Loop and so on.

Background threads want speedy reporting

Assume you want very exact timing, and you can spare an extra cpu core for that. Then you can run an extra thread that signals your perl interpreter. This means you can get a very exact timing source while your perl code is number crunching, without even using a syscall to communicate between your threads.

For example the deliantra game server uses a variant of this technique to interrupt background processes regularly to send map updates to game clients.

Or EV::Loop::Async uses an interrupt object to wake up perl when new events have arrived.

IO::AIO and BDB could also use this to speed up result reporting.

Speedy event loop invocation

One could use this module e.g. in Coro to interrupt a running coro-thread and cause it to enter the event loop.

Or one could bind to `SIGIO` and tell some important sockets to send this signal, causing the event loop to be entered to reduce network latency.

HOW TO USE

You can use this module by creating an `Async::Interrupt` object for each such event source. This object stores a perl and/or a C–level callback that is invoked when the `Async::Interrupt` object gets signalled. It is executed at the next time the perl interpreter is running (i.e. it will interrupt a computation, but not an XS function or a syscall).

You can signal the `Async::Interrupt` object either by calling it's `->signal` method, or, more commonly, by calling a C function. There is also the built-in (POSIX) signal source.

The `->signal_func` returns the address of the C function that is to be called (plus an argument to be used during the call). The signalling function also takes an integer argument in the range SIG_ATOMIC_MIN to SIG_ATOMIC_MAX (guaranteed to allow at least 0..127).

Since this kind of interruption is fast, but can only interrupt a *running* interpreter, there is optional support for signalling a pipe − that means you can also wait for the pipe to become readable (e.g. via EV or AnyEvent). This, of course, incurs the overhead of a `read` and `write` syscall.

USAGE EXAMPLES

Implementing race-free signal handling

This example uses a single event pipe for all signals, and one Async::Interrupt per signal. This code is actually what the AnyEvent module uses itself when Async::Interrupt is available.

First, create the event pipe and hook it into the event loop

```
$SIGPIPE = new Async::Interrupt::EventPipe;
$SIGPIPE_W = AnyEvent->io (
   fh   => $SIGPIPE->fileno,
   poll => "r",
   cb   => \&_signal_check, # defined later
);
```

Then, for each signal to hook, create an Async::Interrupt object. The callback just sets a global variable, as we are only interested in synchronous signals (i.e. when the event loop polls), which is why the pipe draining is not done automatically.

```
my $interrupt = new Async::Interrupt
   cb             => sub { undef $SIGNAL_RECEIVED{$signum} },
   signal         => $signum,
   pipe           => [$SIGPIPE->filenos],
   pipe_autodrain => 0,
;
```

Finally, the I/O callback for the event pipe handles the signals:

```
sub _signal_check {
   # drain the pipe first
   $SIGPIPE->drain;

   # two loops, just to be sure
   while (%SIGNAL_RECEIVED) {
      for (keys %SIGNAL_RECEIVED) {
         delete $SIGNAL_RECEIVED{$_};
         warn "signal $_ received\n";
      }
   }
}
```

Interrupt perl from another thread

This example interrupts the Perl interpreter from another thread, via the XS API. This is used by e.g. the EV::Loop::Async module.

On the Perl level, a new loop object (which contains the thread) is created, by first calling some XS constructor, querying the C−level callback function and feeding that as the `c_cb` into the Async::Interrupt constructor:

```
my $self = XS_thread_constructor;
my ($c_func, $c_arg) = _c_func $self; # return the c callback
my $asy = new Async::Interrupt c_cb => [$c_func, $c_arg];
```

Then the newly created Interrupt object is queried for the signaling function that the newly created thread should call, and this is in turn told to the thread object:

```
_attach $self, $asy->signal_func;
```

So to repeat: first the XS object is created, then it is queried for the callback that should be called when the

Interrupt object gets signalled.

Then the interrupt object is queried for the callback function that the thread should call to signal the Interrupt object, and this callback is then attached to the thread.

You have to be careful that your new thread is not signalling before the signal function was configured, for example by starting the background thread only within `_attach`.

That concludes the Perl part.

The XS part consists of the actual constructor which creates a thread, which is not relevant for this example, and two functions, `_c_func`, which returns the Perl-side callback, and `_attach`, which configures the signalling functioon that is safe toc all from another thread. For simplicity, we will use global variables to store the functions, normally you would somehow attach them to `$self`.

The `c_func` simply returns the address of a static function and arranges for the object pointed to by `$self` to be passed to it, as an integer:

```
void
_c_func (SV *loop)
        PPCODE:
        EXTEND (SP, 2);
        PUSHs (sv_2mortal (newSViv (PTR2IV (c_func))));
        PUSHs (sv_2mortal (newSViv (SvRV (loop))));
```

This would be the callback (since it runs in a normal Perl context, it is permissible to manipulate Perl values):

```
static void
c_func (pTHX_ void *loop_, int value)
{
  SV *loop_object = (SV *)loop_;
  ...
}
```

And this attaches the signalling callback:

```
static void (*my_sig_func) (void *signal_arg, int value);
static void *my_sig_arg;

void
_attach (SV *loop_, IV sig_func, void *sig_arg)
        CODE:
{
        my_sig_func = sig_func;
        my_sig_arg  = sig_arg;

        /* now run the thread */
        thread_create (&u->tid, l_run, 0);
}
```

And `l_run` (the background thread) would eventually call the signaling function:

```
my_sig_func (my_sig_arg, 0);
```

You can have a look at EV::Loop::Async for an actual example using intra-thread communication, locking and so on.

THE Async::Interrupt CLASS

`$async` = new Async::Interrupt key => value...

Creates a new Async::Interrupt object. You may only use async notifications on this object while it exists, so you need to keep a reference to it at all times while it is used.

Optional constructor arguments include (normally you would specify at least one of `cb` or `c_cb`).

cb => `$coderef->`($value)
: Registers a perl callback to be invoked whenever the async interrupt is signalled.

  Note that, since this callback can be invoked at basically any time, it must not modify any well-known global variables such as `$/` without restoring them again before returning.

  The exceptions are `$!` and `$@`, which are saved and restored by Async::Interrupt.

  If the callback should throw an exception, then it will be caught, and `$Async::Interrupt::DIED` will be called with `$@` containing the exception. The default will simply `warn` about the message and continue.

c_cb => [$c_func, `$c_arg`]
: Registers a C callback the be invoked whenever the async interrupt is signalled.

  The C callback must have the following prototype:

```
void c_func (pTHX_ void *c_arg, int value);
```

  Both `$c_func` and `$c_arg` must be specified as integers/IVs, and `$value` is the `value` passed to some earlier call to either `$signal` or the `signal_func` function.

  Note that, because the callback can be invoked at almost any time, you have to be careful at saving and restoring global variables that Perl might use (the exception is `errno`, which is saved and restored by Async::Interrupt). The callback itself runs as part of the perl context, so you can call any perl functions and modify any perl data structures (in which case the requirements set out for `cb` apply as well).

var => `$scalar_ref`
: When specified, then the given argument must be a reference to a scalar. The scalar will be set to `0` initially. Signalling the interrupt object will set it to the passed value, handling the interrupt will reset it to `0` again.

  Note that the only thing you are legally allowed to do is to is to check the variable in a boolean or integer context (e.g. comparing it with a string, or printing it, will *destroy* it and might cause your program to crash or worse).

signal => `$signame_or_value`
: When this parameter is specified, then the Async::Interrupt will hook the given signal, that is, it will effectively call `->signal (0)` each time the given signal is caught by the process.

  Only one async can hook a given signal, and the signal will be restored to defaults when the Async::Interrupt object gets destroyed.

signal_hysteresis => `$boolean`
: Sets the initial signal hysteresis state, see the `signal_hysteresis` method, below.

pipe => [$fileno_or_fh_for_reading, `$fileno_or_fh_for_writing`]
: Specifies two file descriptors (or file handles) that should be signalled whenever the async interrupt is signalled. This means a single octet will be written to it, and before the callback is being invoked, it will be read again. Due to races, it is unlikely but possible that multiple octets are written. It is required that the file handles are both in nonblocking mode.

  The object will keep a reference to the file handles.

  This can be used to ensure that async notifications will interrupt event frameworks as well.

  Note that `Async::Interrupt` will create a suitable signal fd automatically when your program requests one, so you don't have to specify this argument when all you want is an extra file descriptor to watch.

  If you want to share a single event pipe between multiple Async::Interrupt objects, you can use the `Async::Interrupt::EventPipe` class to manage those.

pipe_autodrain => `$boolean`
: Sets the initial autodrain state, see the `pipe_autodrain` method, below.

($signal_func, `$signal_arg`) = `$async->`signal_func
: Returns the address of a function to call asynchronously. The function has the following prototype and needs to be passed the specified `$signal_arg`, which is a `void *` cast to `IV`:

```
void (*signal_func) (void *signal_arg, int value)
```

An example call would look like:

```
signal_func (signal_arg, 0);
```

The function is safe to call from within signal and thread contexts, at any time. The specified `value` is passed to both C and Perl callback.

`$value` must be in the valid range for a `sig_atomic_t`, except `0` (1..127 is portable).

If the function is called while the Async::Interrupt object is already signaled but before the callbacks are being executed, then the stored `value` is either the old or the new one. Due to the asynchronous nature of the code, the `value` can even be passed to two consecutive invocations of the callback.

`$address` = `$async->`c_var
: Returns the address (cast to IV) of an `IV` variable. The variable is set to `0` initially and gets set to the passed value whenever the object gets signalled, and reset to `0` once the interrupt has been handled.

Note that it is often beneficial to just call `PERL_ASYNC_CHECK ()` to handle any interrupts.

Example: call some XS function to store the address, then show C code waiting for it.

```
my_xs_func $async->c_var;

static IV *valuep;

void
my_xs_func (void *addr)
        CODE:
        valuep = (IV *)addr;

// code in a loop, waiting
while (!*valuep)
  ; // do something
```

`$async->`signal ($value=1)
: This signals the given async object from Perl code. Semi-obviously, this will instantly trigger the callback invocation (it does not, as the name might imply, do anything with POSIX signals).

`$value` must be in the valid range for a `sig_atomic_t`, except `0` (1..127 is portable).

`$async->`handle
: Calls the callback if the object is pending.

This method does not need to be called normally, as it will be invoked automatically. However, it can be used to force handling of outstanding interrupts while the object is blocked.

One reason why one might want to do that is when you want to switch from asynchronous interruptions to synchronous one, using e.g. an event loop. To do that, one would first `$async->block` the interrupt object, then register a read watcher on the `pipe_fileno` that calls `$async->handle`.

This disables asynchronous interruptions, but ensures that interrupts are handled by the event loop.

`$async->signal_hysteresis ($enable)`
: Enables or disables signal hysteresis (default: disabled). If a POSIX signal is used as a signal source for the interrupt object, then enabling signal hysteresis causes Async::Interrupt to reset the signal action to `SIG_IGN` in the signal handler and restore it just before handling the interruption.

  When you expect a lot of signals (e.g. when using SIGIO), then enabling signal hysteresis can reduce the number of handler invocations considerably, at the cost of two extra syscalls.

  Note that setting the signal to `SIG_IGN` can have unintended side effects when you fork and exec other programs, as often they do not expect signals to be ignored by default.

`$async->block`
`$async->unblock`
: Sometimes you need a ''critical section'' of code that will not be interrupted by an Async::Interrupt. This can be implemented by calling `$async->block` before the critical section, and `$async->unblock` afterwards.

  Note that there must be exactly one call of `unblock` for every previous call to `block` (i.e. calls can nest).

  Since ensuring this in the presence of exceptions and threads is usually more difficult than you imagine, I recommend using `$async->scoped_block` instead.

`$async->scope_block`
: This call `$async->block` and installs a handler that is called when the current scope is exited (via an exception, by canceling the Coro thread, by calling last/goto etc.).

  This is the recommended (and fastest) way to implement critical sections.

($block_func, `$block_arg`) = `$async->scope_block_func`
: Returns the address of a function that implements the `scope_block` functionality.

  It has the following prototype and needs to be passed the specified `$block_arg`, which is a `void *` cast to `IV`:

```
void (*block_func) (void *block_arg)
```

  An example call would look like:

```
block_func (block_arg);
```

  The function is safe to call only from within the toplevel of a perl XS function and will call `LEAVE` and `ENTER` (in this order!).

`$async->pipe_enable`
`$async->pipe_disable`
: Enable/disable signalling the pipe when the interrupt occurs (default is enabled). Writing to a pipe is relatively expensive, so it can be disabled when you know you are not waiting for it (for example, with EV you could disable the pipe in a check watcher, and enable it in a prepare watcher).

  Note that currently, while `pipe_disable` is in effect, no attempt to read from the pipe will be done when handling events. This might change as soon as I realize why this is a mistake.

`$fileno` = `$async->pipe_fileno`
: Returns the reading side of the signalling pipe. If no signalling pipe is currently attached to the object, it will dynamically create one.

  Note that the only valid operation on this file descriptor is to wait until it is readable. The fd might belong currently to a pipe, a tcp socket, or an eventfd, depending on the platform, and is guaranteed to be `select`able.

`$async->pipe_autodrain ($enable)`
: Enables (`1`) or disables (`0`) automatic draining of the pipe (default: enabled). When automatic draining is enabled, then Async::Interrupt will automatically clear the pipe. Otherwise the user is responsible

for this draining.

This is useful when you want to share one pipe among many Async::Interrupt objects.

`$async->pipe_drain`
Drains the pipe manually, for example, when autodrain is disabled. Does nothing when no pipe is enabled.

`$async->post_fork`
The object will not normally be usable after a fork (as the pipe fd is shared between processes). Calling this method after a fork in the child ensures that the object will work as expected again. It only needs to be called when the async object is used in the child.

This only works when the pipe was created by Async::Interrupt.

Async::Interrupt ensures that the reading file descriptor does not change it's value.

`$signum` = Async::Interrupt::sig2num `$signame_or_number`
`$signame` = Async::Interrupt::sig2name `$signame_or_number`
These two convenience functions simply convert a signal name or number to the corresponding name or number. They are not used by this module and exist just because perl doesn't have a nice way to do this on its own.

They will return `undef` on illegal names or numbers.

THE Async::Interrupt::EventPipe CLASS

Pipes are the predominant utility to make asynchronous signals synchronous. However, pipes are hard to come by: they don't exist on the broken windows platform, and on GNU/Linux systems, you might want to use an `eventfd` instead.

This class creates selectable event pipes in a portable fashion: on windows, it will try to create a tcp socket pair, on GNU/Linux, it will try to create an eventfd and everywhere else it will try to use a normal pipe.

`$epipe` = new Async::Interrupt::EventPipe
This creates and returns an eventpipe object. This object is simply a blessed array reference:

($r_fd, `$w_fd`) = `$epipe->`filenos
Returns the read-side file descriptor and the write-side file descriptor.

Example: pass an eventpipe object as pipe to the Async::Interrupt constructor, and create an AnyEvent watcher for the read side.

```
my $epipe = new Async::Interrupt::EventPipe;
my $asy = new Async::Interrupt pipe => [$epipe->filenos];
my $iow = AnyEvent->io (fh => $epipe->fileno, poll => 'r', cb => sub { });
```

`$r_fd` = `$epipe->`fileno
Return only the reading/listening side.

`$epipe->`signal
Write something to the pipe, in a portable fashion.

`$epipe->`drain
Drain (empty) the pipe.

($c_func, `$c_arg`) = `$epipe->`signal_func
($c_func, `$c_arg`) = `$epipe->`drain_func
These two methods returns a function pointer and `void *` argument that can be called to have the effect of `$epipe->signal` or `$epipe->drain`, respectively, on the XS level.

They both have the following prototype and need to be passed their `$c_arg`, which is a `void *` cast to an `IV`:

```
void (*c_func) (void *c_arg)
```

An example call would look like:

```
        c_func (c_arg);
```

`$epipe->renew`
: Recreates the pipe (usually required in the child after a fork). The reading side will not change it's file descriptor number, but the writing side might.

`$epipe->wait`
: This method blocks the process until there are events on the pipe. This is not a very event-based or ncie way of usign an event pipe, but it can be occasionally useful.

IMPLEMENTATION DETAILS AND LIMITATIONS

This module works by ''hijacking'' SIGKILL, which is guaranteed to always exist, but also cannot be caught, so is always available.

Basically, this module fakes the occurrence of a SIGKILL signal and then intercepts the interpreter handling it. This makes normal signal handling slower (probably unmeasurably, though), but has the advantage of not requiring a special runops function, nor slowing down normal perl execution a bit.

It assumes that `sig_atomic_t`, `int` and `IV` are all async-safe to modify.

AUTHOR

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