Long running opcodes

As Perl interpreter only looks at the signal flags when it about to execute a new opcode if a signal arrives during a long running opcode (e.g. a regular expression operation on a very large string) then signal will not be seen until operation completes.

Interrupting IO

When a signal is delivered (e.g. INT control-C) the operating system breaks into IO operations like `read` (used to implement Perls <> operator). On older Perls the handler was called immediately (and as `read` is not "unsafe" this worked well). With the "deferred" scheme the handler is not called immediately, and if Perl is using system's `stdio` library that library may re-start the `read` without returning to Perl and giving it a chance to call the %SIG handler. If this happens on your system the solution is to use `:perlio` layer to do IO - at least on those handles which you want to be able to break into with signals. (The `:perlio` layer checks the signal flags and calls %SIG handlers before resuming IO operation.)

Note that the default in Perl 5.7.3 and later is to automatically use the `:perlio` layer.

Note that some networking library functions like gethostbyname() are known to have their own implementations of timeouts which may conflict with your timeouts. If you are having problems with such functions, you can try using the POSIX sigaction() function, which bypasses the Perl safe signals (note that this means subjecting yourself to possible memory corruption, as described above). Instead of setting `$SIG{ALRM}` try something like the following:

```
use POSIX;
sigaction SIGALRM, new POSIX::SigAction sub { die "alarm\n" }
    or die "Error setting SIGALRM handler: $!\n";
```

Restartable system calls

On systems that supported it, older versions of Perl used the SA_RESTART flag when installing %SIG handlers. This meant that restartable system calls would continue rather than returning when a signal arrived. In order to deliver deferred signals promptly, Perl 5.7.3 and later do *not* use SA_RESTART. Consequently, restartable system calls can fail (with $! set to `EINTR`) in places where they previously would have succeeded.

Note that the default `:perlio` layer will retry `read`, `write` and `close` as described above and that interrupted `wait` and `waitpid` calls will always be retried.

Signals as "faults"

Certain signals e.g. SEGV, ILL, BUS are generated as a result of virtual memory or other "faults". These are normally fatal and there is little a Perl-level handler can do with them. (In particular the old signal scheme was particularly unsafe in such cases.) However if a %SIG handler is set the new scheme simply sets a flag and returns as described above. This may cause the operating system to try the offending machine instruction again and - as nothing has changed - it will generate the signal again. The result of this is a rather odd "loop". In future Perl's signal mechanism may be changed to avoid this - perhaps by simply disallowing %SIG handlers on signals of that type. Until then the work-round is not to set a %SIG handler on those signals. (Which signals they are is operating system dependant.)

Signals triggered by operating system state

On some operating systems certain signal handlers are supposed to "do something" before returning. One example can be CHLD or CLD which indicates a child process has completed. On some operating systems the signal handler is expected to `wait` for the completed child process. On such systems the deferred signal scheme will not work for those signals (it does not do the `wait`). Again the failure will look like a loop as the operating system will re-issue the signal as there are un-waited-for completed child processes.

If you want the old signal behaviour back regardless of possible memory corruption, set the environment variable `PERL_SIGNALS` to `"unsafe"` (a new feature since Perl 5.8.1).