Add a `timeout` option to `IO.copy_stream`

09/27/2021 12:16 PM - byroot (Jean Boussier)

**Status:** Open
**Priority:** Normal
**Assignee:**
**Target version:**

**Description**

**Context**

In many situations dealing with large files, IO.copy_stream when usable bring major performance gains (often twice faster at the very least). And more importantly, when the copying is deferred to the kernel, the performance is much more consistent as it is less impacted by the CPU utilization on the machine.

However, it is often unsafe to use because it doesn't have a timeout, so you can only use it if both the source and destination IOs are trusted, otherwise it is trivial for an attacker to DOS the service by reading the response very slowly.

**Some examples**

- It is used by webrick.
- Net::HTTP uses it to send request body if they are IOs, but it is used with a "fake IO" to allow for timeouts, so sendfile(2) &co are never used.
- A proof of concept of integrating in puma shows a 2x speedup.
- Various other HTTP client could use it as well.
- I used it in private projects to download and upload large archives in and out of Google Cloud Storage with great effects.

**Possible implementation**

The main difficulty is that the underlying sycalls don't have a timeout either.

The main syscall used in these scenarios is sendfile(2). It doesn't have a timeout parameter, however if called on file descriptors with O_NONBLOCK it does return early and allow for a select/poll loop. I did a very quick and dirty experiment with this, and it does seem to work.

The other two accelerating sycalls are copy_file_range(2) (linux) and fcopyfile(2) (macOS). Neither have a timeout, and neither manpage document an EAGAIN / EWOULDBLOCK error. However these sycalls are limited to real file copies, generally speaking timeouts for real files are less of a critical need, so it would be possible to simply not use these sycalls if a timeout is provided.

**Interface**

`copy_stream(src, dst, copy_length, src_offset, timeout)`

or `copy_stream(src, dst, copy_length, src_offset, timeout: nil)`

As for the return value in case of a timeout, it is important to convey both that a timeout happened, and the number of bytes that were copied, otherwise it makes retries impossible.

- It could simply returns the number of byte, and let the caller compare it to the expected number of bytes copied, but that wouldn't work in cases where the size of src isn't known.
- It could return -1 - bytes_copied, not particularly elegant but would work.
- It could return multiple values or some kind of result object when a timeout is provided.
- It could raise an error, with bytes_copied as an attribute on the error.

Or alternatively copy_stream would be left without a timeout, and some kind of copy_stream2 would be introduced so that copy_stream return value wouldn't be made inconsistent.

**Related issues:**

Related to Ruby master - Bug #17933: `Net::HTTP#write_timeout` doesn't work w... Closed

**History**

#1 - 09/27/2021 12:16 PM - byroot (Jean Boussier)
I wonder, can sendfile(2) be interrupted by a signal like SIGVTALRM like for read/write? That might be another strategy to implement a timeout for it.

Just FYI: sendfile is less flexible and you should generally avoid it. The modern syscall is splice. I'll read the rest of the issue in more detail and reply later.

I think it can? But from my initial research I was under the impression that using ALARM to interrupt system calls was a bit frowned upon and that poll/select was generally regarded as preferable.

Indeed. IO.copy_stream could use some more modern syscalls like splice and io_uring. But sendfile(2) has the advantage to be relatively portable (older linux, macOS, freeBSD, etc).

So to me this is a bit orthogonal.

It's what Ruby already uses to interrupt blocking system calls for Ruby interrupts like signals, Thread#raise, etc (as you might already know), so I think it's fine.

Making an IO non-blocking for IO.copy_stream might have side effects, but maybe they are fine.

Once a timeout occurs, are you sure we can make such a guarantee about the number of bytes that are copied reliably?

Depends. In sendfile(2) case, if we use O_NONBLOCK then yes we can trust the return value:

If the transfer was successful, the number of bytes written to out_fd is returned. Note that a successful call to sendfile() may write fewer bytes than requested; the caller should be prepared to retry the call if there were unsent bytes.
And I believe it's the same for splice(2).

However if we use a signal to interrupt it (assuming it's possible, because the man file doesn't document EINTR so I'm not so sure), then the return value would be -1 so we wouldn't have this information.

#9 - 09/30/2021 08:06 PM - Eregon (Benoit Daloze)

byroot (Jean Boussier) wrote in #note-8:

However if we use a signal to interrupt it (assuming it's possible, because the man file doesn't document EINTR so I'm not so sure), then the return value would be -1 so we wouldn't have this information.

I would think it also returns the number of written bytes, otherwise that syscall is completely unusable in the presence of signals (it could infinitely loop with retries).

See write(2) for instance, it returns the number of written bytes:

Note that a successful write() may transfer fewer than count bytes. Such partial writes can occur for various reasons; for example, because there was insufficient space on the disk device to write all of the requested bytes, or because a blocked write() to a socket, pipe, or similar was interrupted by a signal handler after it had transferred some, but before it had transferred all of the requested bytes.

#10 - 10/01/2021 04:49 AM - normalperson (Eric Wong)

"Eregon (Benoit Daloze)" noreply@ruby-lang.org wrote:

I wonder, can sendfile(2) be interrupted by a signal like SIGVTALRM like for read/write?
That might be another strategy to implement a timeout for it.

For the socket/pipe ends, yes (if blocking), that is interruptible. Not for regular files (unless it's something like NFS mounted with interrupts enabled).

Feature #18228: Add a timeout option to IO.copy_stream
https://bugs.ruby-lang.org/issues/18228#change-93902

#11 - 10/01/2021 05:10 AM - normalperson (Eric Wong)

"ioquatix (Samuel Williams)" noreply@ruby-lang.org wrote:

Just FYI: sendfile is less flexible and you should generally avoid it. The modern syscall is splice.

No point in avoiding sendfile, sendfile is considerably easier to use in common cases and results in fewer syscalls.

splice requires one end to be a pipe; if neither end is a pipe so you need to create and manage the pipe yourself as an intermediate buffer. Basically, instead of:

void *buf = malloc(...);
while (read(rfd, buf, ...) > 0)
write(wfd, buf, ...);
free(buf);

It becomes:

int buf[2];
pipe2(buf, ...);
while (splice(rfd, ..., buf[1], ...) > 0) /* splice into pipe */
splice(buf[0], ..., wfd, ...); /* splice out of pipe */
close(buf[0]);
close(buf[1]);

sendfile creates an internal pipe transparently inside the
kernel for doing splice. The pipe is still there, but private
to the kernel so you won't have to jump between userspace and
kernel space repeatedly.