Ruby master - Feature #17059

epoll as the backend of IO.select on Linux

07/29/2020 03:33 PM - dsh0416 (Delton Ding)

Status:	Rejected
Priority:	Normal
Assignee:	
Target version:	

Description

Current Ruby's IO.select method calls POSIX select API directly. With the new non-blocking scheduler, this may be the bottleneck of the I/O scheduling. For keeping the backward compatibilty of the current IO.select methods, a proposal may be to create a "duck" select which uses the epoll_wait as the backend.

One tricky part is that the fd_set described in POSIX is write-only, which means it is impossible to iterate for generating the epoll_event argument for epoll_wait. But similar to the large-size select situation, we could define our own rb_fdset_t struct in this case, and implement the following APIs.

```
void rb_fd_init(rb_fdset_t *);
void rb_fd_term(rb_fdset_t *);
void rb_fd_zero(rb_fdset_t *);
void rb_fd_set(int, rb_fdset_t *);
void rb_fd_clr(int, rb_fdset_t *);
int rb_fd_isset(int, const rb_fdset_t *);
void rb_fd_copy(rb_fdset_t *, const fd_set *, int);
void rb_fd_dup(rb_fdset_t *dst, const rb_fdset_t *src);
int rb_fd_select(int, rb_fdset_t *, rb_fdset_t *, rb_fdset_t *, struct timeval *);
```

TODO:

1. Implement the fd_set with dynamic allocated fds.
2. Implement the epoll with select API.
3. Edit io.c to use the customized fd_set struct.

I'm trying to work on a branch for this. Any suggestions for this?

History

#1 - 07/29/2020 03:34 PM - dsh0416 (Delton Ding)

- Subject changed from epoll as IO.select to epoll as IO.select's backend

#2 - 07/29/2020 03:41 PM - dsh0416 (Delton Ding)

- Description updated

#3 - 07/29/2020 03:42 PM - dsh0416 (Delton Ding)

- Subject changed from epoll as IO.select's backend to epoll as the backend of IO.select

#4 - 07/29/2020 03:46 PM - dsh0416 (Delton Ding)

- Subject changed from epoll as the backend of IO.select to epoll as the backend of IO.select on Linux

#5 - 07/29/2020 03:55 PM - dsh0416 (Delton Ding)

- File epoll.h added

#6 - 08/12/2020 07:38 AM - ko1 (Koichi Sasada)

does it improve the performance?
My understanding is the advantage of epoll is advanced registration.
However, select wrapping interface can not use it.

I have no experience to use epoll, so correct me it is wrong.

Thanks,

#7 - 08/16/2020 01:12 AM - dsh0416 (Delton Ding)

It should greatly improve the performance.
Advanced registration is a feature of epoll,
but the performance is also an important part for it.
The benchmark from libevent shows the performance of epoll and poll or select,
are on a totally different stage.
So this may be important to performance.

#8 - 08/16/2020 09:02 AM - shyouhei (Shyouhei Urabe)

ko1 (Koichi Sasada) is not talking about efficiency of epoll in general, but questioning that of your patch.

#9 - 08/16/2020 10:24 AM - dsh0416 (Delton Ding)

In general, event handling gems like nio4r could provide a similar select interface with multiple backends including select, kqueue and epoll support.

On the side of Ruby meta-programming, this part is easy to be implemented, and could provide a much better performance comparing to the default IO.select.

Since Ruby merged the Fiber scheduler recently, the IO.select injection from the core library may be important to provide better native I/O performance.

But Ruby's implementation of the IO.select has a lot of things coupled with the POSIX select, like the fdset_t.

From the patch, Ruby does use the POSIX fdset_t in some platforms, but Ruby also defines its own structs on some other platforms for non-standard select implementation. this gives the opportunity to inject the epoll API here.

This is my current idea, and I'm working on implementing this.

I'm wondering if changing the rb_f_select directly could be a better idea or not. But there's no such macro control yet in this method, and all other implementations are with the select_internal.

#10 - 08/16/2020 09:08 PM - dsh0416 (Delton Ding)

- File epoll.h added

Update the WIP implementation

#11 - 08/16/2020 11:49 PM - ioquatix (Samuel Williams)

This is a nice idea, and I've considered exposing wait_select on the scheduler interface.

I'd suggest we try to go down this route.

The scheduler interface can cache the I/O registration... so in theory it addresses ko1 (Koichi Sasada)'s concerns.

#12 - 08/17/2020 07:37 AM - dsh0416 (Delton Ding)

Thanks for advice.

To separate the process of registration and wait is a good idea for performance.

Since even the select itself could also take advantages from this,

and simplify the whole I/O multiplexing process.

#13 - 08/17/2020 09:40 AM - Eregon (Benoit Daloze)

I'm unclear how using epoll can help for the user calling IO.select.
With the API of select(), I'd expect epoll is no faster than select().

Regarding the Fiber scheduler, then I think a new event would be needed, calling epoll() instead of select() wouldn't solve anything, right?

#14 - 08/17/2020 09:49 AM - ko1 (Koichi Sasada)

I want to know the general idea how to use epoll for IO.select backend.

```
#include <stdlib.h>
#include <stdio.h>

#define _GNU_SOURCE
#include <unistd.h>
```

```
#include <sys/resource.h>
#include <poll.h>
#define N 2000 // 10k

static void
task_poll(int fds[])
{
    struct pollfd pfd[N];
    for (int i=0; i<N; i++) {
        struct pollfd *p;
        // in
        p = &pfd[i];
        p->fd = fds[i*2];
        p->events = POLLIN;
    }

    int r = poll(&pfd[0], N, 0);
    if (r==0) {
        // timeout
    }
    else if (r>0) {
        for (int i=0; i<N*2; i++) {
            fprintf(stderr, "%d %d (%d)\n", i, pfd[i].fd, (int)pfd[i].revents);
        }
    }
    else {
        fprintf(stderr, "poll (RLIMIT_NOFILE:%d and N:%d)\n", (int)RLIMIT_NOFILE, N*2);
        exit(1);
    }
}

#include <sys/epoll.h>

static void
task_epoll(int fds[])
{
    struct epoll_event events[N];
    int efd = epoll_create(N);
    if (efd < 0) {
        perror("epoll");
        exit(1);
    }

    for (int i=0; i<N; i++) {
        struct epoll_event *e = &events[i];
        e->events = EPOLLIN;
        e->data.fd = fds[i*2];
        if (epoll_ctl(efd, EPOLL_CTL_ADD, fds[i*2], e) < 0) {
            perror("epoll_ctl");
            exit(1);
        }
    }

    int r = epoll_wait(efd, events, N, 0);
    if (r == 0) {
        // timeout
    }
    else if (r > 0) {
        for (int i=0; i<r; i++) {
            fprintf(stderr, "%d fd:%d\n", i, events[i].data.fd);
        }
    }
    else {
        perror("epoll_wait");
        exit(1);
    }

    // clear
    close(efd);
}


int main(void)
{
```

```
    int fds[N * 2];
    int i;
    for (i=0; i<N; i++) {
        if (pipe(&fds[i*2]) < 0) {
            perror("pipe");
            fprintf(stderr, "i:%d\n", i);
            exit(1);
        }
    }

    for (i=0; i<1000 * 10; i++) {
        // task_xxx emulates IO.select

        // task_poll(fds);  // real    0m0.537s
        // task_epoll(fds); // real    0m11.191s
    }

    return 0;
}
```

epoll version is x20 slower on my machine.
any misunderstanding?

(efd can be reusable, but I'm not sure how to clear all registered fds)

#15 - 08/17/2020 09:53 AM - ko1 (Koichi Sasada)

I understand epoll will help by introducing new event handling APIs.
But not sure why IO.select can improve the performance with epoll.
This is my question by first comment and maybe https://bugs.ruby-lang.org/issues/17059#note-13 is same.

#16 - 08/17/2020 12:00 PM - dsh0416 (Delton Ding)

The benchmark looks good. I've tested with similar code, and it's 46x slower on my machine.
It looks like epoll is highly depended on the time that epoll_ctl engaged.

Since the scheduler now have other registration control including rb_io_wait_readable and rb_io_wait_writable are introduced in the current Scheduler.
I would try to use these methods to deal with the registration then, and replace the IO.select in the Scheduler#run for performance.

Is this a proper way to implement then?

#17 - 08/17/2020 05:37 PM - ko1 (Koichi Sasada)

- Status changed from Open to Rejected

> Since the scheduler now have other registration control including rb_io_wait_readable and rb_io_wait_writable are introduced in the current Scheduler.
> I would try to use these methods to deal with the registration then, and replace the IO.select in the Scheduler#run for performance.

I'm not sure what is your idea, but at least I reject this ticket because IO.select is not good place to use epoll.

Maybe design Ruby-level event handling API and use it by scheduler is the best.

```
s = IO::Selector
s.add(io1, 'r')
s.add(io2, 'w')
s.add(io3, 'rw')
s.wait #=> ready io array
```

for example.

#18 - 08/18/2020 09:07 AM - Eregon (Benoit Daloze)

dsh0416 (Delton Ding) wrote in #note-16:

> I would try to use these methods to deal with the registration then, and replace the IO.select in the Scheduler#run for performance.

Where do you see IO.select in the Scheduler?
Here? https://github.com/ruby/ruby/blob/701217572f865375b137d2830d4da0c3e78de046/test/fiber/scheduler.rb#L44
That's a test scheduler, and using IO.select() is good enough for prototyping but basically nothing else.

ioquatix (Samuel Williams) it seems worth clarifying that with a comment there.
Might also be worth linking to the async scheduler as a real-world example there or in doc/fiber.rdoc.

Real schedulers like the one for async use epoll/kqueue/libev/etc internally, for instance by using nio4r.

#19 - 08/18/2020 09:09 AM - Eregon (Benoit Daloze)

In other words I don't think we need to have selectors in Ruby core.
That's part of the beauty of this new scheduler API: it lets you implement it in various way, including how do you want to wait for IO.

#20 - 08/18/2020 04:05 PM - dsh0416 (Delton Ding)

Yes. I was just figured out that the scheduler is an example in the tests, where the real scheduler is designed to be separated from the ruby-core.

Files

epoll.h	3.62 KB	07/29/2020	dsh0416 (Delton Ding)
epoll.h	6.44 KB	08/16/2020	dsh0416 (Delton Ding)