Unix **connect()** and interrupted system calls

Summary: This page makes a fine (and admittedly minor) point about the behavior of the Unix (socket API) `connect()` system call when it is interrupted by a signal. It points out how obscure the Single Unix Specification is, and notes that many existing Unix implementations seem to have f*cked this up somehow.

The question is this: if a blocking `connect()` (on a blocking stream socket, that is) is interrupted by a signal, returning `EINTR`, in what state is the socket left, and is it permissible to restart the system call? What happens if a second `connect()` with the same arguments is attempted immediately after one failed with `EINTR`?

The reference for `connect()` (hereafter, “the Spec”) is part of the Open Group's Single Unix Specification, version 3 (**note:** you may need to register to read this; see also here). Here is the relevant part of it:

> If the initiating socket is connection-mode, then `connect()` shall attempt to establish a connection to the address specified by the *address* argument. If the connection cannot be established immediately and `O_NONBLOCK` is not set for the file descriptor for the socket, `connect()` shall block for up to an unspecified timeout interval until the connection is established. If the timeout interval expires before the connection is established, `connect()` shall fail and the connection attempt shall be aborted. If `connect()` is interrupted by a signal that is caught while blocked waiting to establish a connection, `connect()` shall fail and set `connect()` to [`EINTR`], but the connection request shall not be aborted, and the connection shall be established asynchronously.
>
> If the connection cannot be established immediately and `O_NONBLOCK` is set for the file descriptor for the socket, `connect()` shall fail and set `errno` to [`EINPROGRESS`], but the connection request shall not be aborted, and the connection shall be established asynchronously. Subsequent calls to `connect()` for the same socket, before the connection is established, shall fail and set `errno` to [`EALREADY`].
>
> When the connection has been established asynchronously, `select()` and `poll()` shall indicate that the file descriptor for the socket is ready for writing.

Later on, when listing possible error codes for `connect()`, the Spec mentions:

> The `connect()` function shall fail if:
>
> […]
> `[EALREADY]`
> A connection request is already in progress for the specified socket.

How does this answer the question above? That is, what is supposed to happen, according to the Spec, when a second `connect()` is attempted (with the same arguments) just after one which returned `EINTR` (on a blocking stream socket)?

To me it seems that the Spec is contradictory. On the one hand, it is stated that “If the connection cannot be established immediately and `O_NONBLOCK` is not set for the file descriptor for the socket,

`connect()` shall block for up to an unspecified timeout interval until the connection is established."—now that sort of implies, since no provision is made for exceptions, that the second call to `connect()` should *continue* the connection attempted by the first (and which has not been aborted but run asynchronously), and block until the connection is established (or an error is returned). Let us call this the "Liberal Behavior" in what follows (I will later explain that this is how Linux behaves). On the other hand, the `EALREADY` error is documented as being returned whenever a connection request is underway. So the Spec seems to say that the second call to `connect()` should fail with the `EALREADY` error code. Let us call this the "Unforgiving Behavior" in what follows (I will later explain that this is how Solaris behaves). One thing is certain: the Spec is highly unclear about this point.

I have asked several people's opinion as to how they read the Spec, and most seem to favor the "Unforgiving Behavior": they say the Spec requires the second `connect()` call to fail with `EALREADY`. So admittedly it is how the Spec should be read; whether this is *desirable* behavior, on the other hand, is dubious (see below).

In *Unix Network Programming*, volume 1, section 5.9, W. Richard Stevens states:

> What we are doing […] is restarting the interrupted system call ourself. This is fine for `accept`, along with the functions such as `read`, `write`, `select` and `open`. But there is one function that we cannot restart ourself: `connect`. If this function returns `EINTR`, we cannot call it again, as doing so will return an immediate error. When `connect` is interrupted by a caught signal and is not automatically restarted, we must call `select` to wait for the connection to complete, as we describe in section 15.3.

This, indeed, clearly describes the "Unforgiving Behavior" (`connect()` failing immediately when restarted); note that Stevens does not say which error code is produced, and indeed Solaris returns `EALREADY` but BSD returns `EADDRINUSE` (a highly illogical error code in my opinion).

The "Liberal Behavior" consists of making `connect()` like every other system call: it can be restarted with the same arguments, without thinking, so long as it returns `EINTR` (only one minor difference remains: `EISCONN` needs to be checked, to avoid a race condition between two calls to `connect()`). This seems to be what Linux does (whether this is against the Spec, is, as I say, a matter of interpretation, though most people seem to think that indeed it is). There is much to be said in favor of this "Liberal Behavior": basically, *the whole point of using blocking sockets is for system calls to block* rather than stupidly returning a temporary error code (be it `EWOULDBLOCK`/`EAGAIN`, `EINPROGRESS` or `EALREADY`) and forcing us to use `select()` or `poll()` to know when that temporary error will be gone; of what use are blocking sockets at all, if we are forced to use `select()` or `poll()` anyway? This is my main reason for preferring the "Liberal Behavior".

Besides, the "Liberal Behavior" makes things *much* easier to program. In clear, I can write the following:

```
/* Start with fd just returned by socket(), blocking, SOCK_STREAM... */
while ( connect (fd, &name, namelen) == -1 && errno != EISCONN )
  if ( errno != EINTR )
    {
      perror ("connect");
      exit (EXIT_FAILURE);
    }
/* At this point, fd is connected. */
```

—instead of having to write all this:

```
/* Start with fd just returned by socket(), blocking, SOCK_STREAM... */
if ( connect (fd, &name, namelen) == -1 )
  {
    struct pollfd unix_really_sucks;
    int some_more_junk;
```

```
        socklen_t yet_more_useless_junk;

        if ( errno != EINTR /* && errno != EINPROGRESS */ )
          {
            perror ("connect");
            exit (EXIT_FAILURE);
          }
        unix_really_sucks.fd = fd;
        unix_really_sucks.events = POLLOUT;
        while ( poll (&unix_really_sucks, 1, -1) == -1 )
          if ( errno != EINTR )
            {
              perror ("poll");
              exit (EXIT_FAILURE);
            }
        yet_more_useless_junk = sizeof(some_more_junk);
        if ( getsockopt (fd, SOL_SOCKET, SO_ERROR,
                         &some_more_junk,
                         &yet_more_useless_junk) == -1 )
          {
            perror ("getsockopt");
            exit (EXIT_FAILURE);
          }
        if ( some_more_junk != 0 )
          {
            fprintf (stderr, "connect: %s\n",
                     strerror (some_more_junk));
            exit (EXIT_FAILURE);
          }
      }
    /* At this point, fd is connected. */
```

—which anyone will admit is longer (over five times longer, as a matter of fact) and more tedious to write. Hence my calling this behavior the “Liberal Behavior” because it does not force the programmer to go through all this pain just to connect a socket.

Unfortunately, my opinion has not been consulted in defining Unix implementations, nor in writing the Single Unix specification, so it seems that the “Liberal Behavior” is not highly thought of, except under Linux, and anyone who wants to write a Unix program (except if it is to run solely on the Linux kernel) that performs the trivial act of opening a socket has to go through all the mess I have just written (and which, in case it is of any use, I put in the Public Domain). Why not use `SA_RESTART` on all signal handlers, some will ask? Well, the problem is that one is never quite sure that *all* signals have been treated, and just one signal is enough to interrupt system calls. Most people agree that `SA_RESTART` does not dispense you from testing `EINTR` everywhere, if you want to be safe. Anyway, it is not the purpose of this page to descuss this point.

Annoyingly, not only Unix implementations vary in this, but also the documentation is either imprecise or positively wrong.

Linux adopts the pleasant “Liberal Behavior” I have described. The man page for `connect(2)` under Linux does not document `EINTR`, however, nor does the info page of the GNU libc, whereas the code can be shown to occur: `connect()` *can* be interrupted by a signal under Linux (fortunately!).

Solaris adopts the “Unforgiving Behavior” that seems to be the literal interpretation of the Spec. However, the Solaris man page for `connect(3SOCKET)` states that the `EALREADY` error code occurs when “The socket is non-blocking and a previous connection attempt has not yet been completed.” But I have cases when the `EALREADY` error code was returned for a blocking socket (this is the very point I'm arguing about).

Both FreeBSD and OpenBSD adopt the “Unforgiving Behavior” with the following departure from the Spec that the error code returned on the second `connect()` call is `EADDRINUSE` rather than `EALREADY`. This is probably tradition, but it seems rather absurd. There is, however, one difference between FreeBSD and OpenBSD, but it concerns *non-blocking* sockets: OpenBSD returns `EALREADY` for non-blocking sockets as the Spec prescribes, whereas FreeBSD never seems to return `EALREADY` at all. However, this page is essentially about blocking sockets. Also, FreeBSD documents `EALREADY` (strange, since it does not return it), but not `EINTR` (which it does return); the OpenBSD man page is essentially correct. Note that in both cases the return code of `EADDRINUSE` is not clearly documented.

Information about the behavior of other Unixen, or other implementations of the Unix socket API, in this regard, is welcome.

See also this thread on the `comp.unix.programmer` newsgroup (thanks to Google Groups for making this available), where the issue was discussed (note that I mentioned `EISCONN` in error whereas I meant `EADDRINUSE`).

This trivial program (Public Domain) will run tests on a given Unix implementation, to determine what the actual behavior happens to be, and display the result in a perfectly obscure and incomprehensible form. See the comments at the beginning of the file for information on use.

David Madore

Last modified: $Date: 2003/04/25 04:22:23 $