A history of the fd_set, FD_SETSIZE, and how it relates to WinSock

devblogs.microsoft.com/oldnewthing/20221102-00

November 2, 2022

Raymond Chen

The `select` function takes three sets of file descriptors (`readfds` , `writefds` , `exceptfds`) and waits for any of the following things to happen:

- A file descriptor in `readfds` is available to read without blocking. (Usually this means that there is data available.)
- A file descriptor in `writefds` is available to write without blocking. (Usually this means that there is buffer space available.)
- A file descriptor in `exceptfds` is in an error state.
- The call times out, if a timeout is provided.

The function signature is

```
int select(int nfds, fd_set *readfds,
    fd_set *writefds, fd_set *exceptfds,
    struct timeval *timeout);
```

where `nfds` is one greater than the highest-numbered file descriptor in any of the `fd_set` s.

When originally implemented, the ABI representation of `fd_set` was a bit array consisting of `nfds` bits.[1] The bits are numbered starting with zero, and the file descriptors you want to be part of the `fd_set` have the corresponding bits set in the array.

For example, if you wanted the `fd_set` to represent file descriptors 4 and 6, you would use a bit array whose first byte was `(1 << 4) | (1 << 6) = 0x50` .

The `sys/select.h` header file provided a stock implementation of `fd_set` : The stock implementation represented a a bit array of `FD_SETSIZE` bits. You could configure the header file by defining the `FD_SETSIZE` preprocessor symbol to the number of bits you desire, and if you didn't customize the value, it defaulted to 1024. (We'll see why later.)

The physical representation of the `fd_set` was hidden behind macros, and you were expected to use those macros rather than trying to build the bitmap yourself.

```
FD_ZERO(set)        // set all bits in the fd_set to zero
FD_SET(set, fd)     // set bit "fd" in the fd_set to 1
FD_CLR(set, fd)     // set bit "fd" in the fd_set to 0
FD_ISSET(set, fd)   // returns nonzero if bit "fd" is set
```

Picking a value for `FD_SETSIZE` size is a bit of a balancing act. You don't want the bitmap to be too big, because that's a lot of memory for each `fd_set` , and the `select` function is going to have to scan that many bits to find the ones that are set. On the other hand, you need it to be big enough to handle all reasonable workloads.

The operating system itself imposes a limit on the size of the bitmap, because large bitmaps require a lot of memory and are slow to scan for set bits.[2] At the time that `select` was invented, each process was limited to 1024 simultaneous open files, which seemed like a generous allowance at the time. I mean, who's going to write a program that opens over a thousand sockets at once?

Since you could have at most 1024 open file descriptors at a time, file descriptors were in practice always in the range 0…1023, and therefore a default value for `FD_SETSIZE` of 1024 would be sufficient to cover all the file descriptors a program could possibly open.

Although the `fd_set` was in theory an opaque data type, it was effectively immutable because the management of the `fd_set` was done by macros, so all of the manipulations were hard-coded into every program that used `fd_set` s. And of course the entire thing was a bit array at the ABI layer, and you don't want to break the ABI.

The 1024-file descriptor limit became a problem as computers became larger and more powerful, and the choice of representation as a bit array meant that just increasing the bit array size limit was not a good solution. If you wanted to wait for file descriptors 4000 and 8000, you'd have to allocate three kilobyte-sized bit arrays. That's three big bit arrays just to set two bits.

At this point, the history splits into two story lines.

On the Unix side, instead of making the bit arrays larger and larger, POSIX invented `poll` , a replacement for `select` which expresses the file descriptor set differently. Instead of being a bit array whose size is proportional to the total number of file descriptors the process has opened, it's an array of `pollfd` structures, each of which describes a file descriptor you want to wait for.

This brings the size and complexity of the operation down to a reasonable level, since the amount of memory that is needed now grows with the number of file descriptors you are monitoring, rather than with the total number of file descriptors in the process. (Linux, meanwhile, added their own extension to `poll` called `epoll` .)

The `select` function never gained the ability to pass bit arrays larger than 1024 bits. The functionality got frozen at its old level of support, with the expectation that people who need to manage thousands of file descriptors will move to `poll`.

Meanwhile, WinSock wanted to maintain source compatiblity with existing Unix-based networking code, including code that used `select`. On the other hand, the `fd_set` design didn't fit well with Windows, because Windows sockets are handles (arbitrary pointer-sized values), not guaranteed to be small integers. Fortunately, WinSock could invent its own ABI, since there were no pre-existing binaries to be compatible with. The ABI for `fd_set` in WinSock takes a different form:

```
struct fd_set_abi
{
    uint32_t count;
    SOCKET   sockets[count]; // variable-length array
};
```

The WinSock ABI for the `fd_set` consists of a count (specifying how many sockets are in the set), followed by an array of that many socket handles.

The various `FD_` macros were redefined so that they preserved the programming interface but operated on this alternate representation on an `fd_set`.

This alternate design avoids tying the bit array to the numeric values of the socket handles, which could in theory force an array larger than available memory: If a handle happened to be `0xdeadbeef`, that would require nearly a half-gigabyte-sized bit array just so you could set the 3 billionth bit (and leave all the other bits zero). It also avoids the problem of having to validate and capture very large bit arrays at the kernel boundary.

As with the original `sys/select.h`, you can define the preprocessor symbol `FD_SETSIZE` before including `winsock.h`, and that changes the size of the default implementation of `fd_set` as well as the behavior of the corresponding `fd_set`-manipulation macros.

A caveat applies to both the Unix-style and the WinSock-style `fd_set` structures: Since the definition changes depending on how you set `FD_SETSIZE`, a program which defines `FD_SETSIZE` differently in different translation units is technically in violation of the C++ standard One Definition Rule, because it results in different definitions of `fd_set` in different translation units. In practice, you usually get away with it if you pass the `fd_set` objects only by address or by reference. Still, bad things will happen if one function creates an `fd_set` with one `FD_SETSIZE`, and passes it to another function that was compiled with a different `FD_SETSIZE`, and the recipient function tries to modify the `fd_set`, because the recipient will operate on the assumption that the `fd_set` can hold `FD_SETSIZE` entries, for a different value of `FD_SETSIZE`.

For Windows, you can define your own ABI-compatible version of `fd_set` that encodes its own maximum size:

```
// Warning: Works only for WinSock fd_set.

template<uint32_t Capacity = FD_SETSIZE>
struct fd_setN
{
    uint32_t count = 0;
    std::array<SOCKET, Capacity> array;

    operator fd_set*() noexcept { return reinterpret_cast<fd_set*>(this); }

    constexpr auto capacity() const noexcept { return Capacity; }
    auto begin() noexcept { return &array[0]; }
    auto end() noexcept { return &array[count]; }

    bool add(SOCKET s) noexcept
    {
        if (count >= capacity()) return false;
        array[count++] = s;
        return true;
    }

    bool remove(SOCKET s) noexcept
    {
        auto it = std::find(begin(), end(), s);
        if (it == end()) return false;
        *it = array[--count];
        return true;
    }

    bool contains(SOCKET s) noexcept const
    {
        return std::find(begin(), end(), s) != end();
    }

    bool clear() noexcept
    {
        count = 0;
    }
};
```

You are welcome to add your own methods to this class, like say `push_back` to make it satisfy more of the C++ Container requirements,[1] but this is the basic idea.

Of course, passing large arrays to `select` generates a lot of busy-work both for you and the operating system: You have to go into a loop filling the `fd_set` before each call to `select` (because `select` modifies the `fd_set` before returning), and the operating system has to

go into a loop to set up a monitor to track the status of each socket, and then another loop to cancel the monitor once one of them becomes ready.

A better model for this is to use `WSASelectEvent` to associate an event with each socket. This association is independent of activity on other sockets, so you don't have to keep re-establishing it.

Even better would be to use an I/O completion port to deal with each socket's readiness.

Exercise: Armed with this information, maybe you can help this customer, who reported that `FD_ISSET` is unreliable on Windows:

```
struct socket_manager
{
    fd_set sockets;

    socket_manager()
    {
        FD_ZERO(&sockets);
    }

    void add(int socket)
    {
        FD_SET(socket, &sockets);
    }

    bool select()
    {
        fd_set ready = sockets;
        if (select(FD_SETSIZE, &ready,
                   nullptr, nullptr, nullptr) < 0) {
            return false;
        }
        // Process each ready socket.
        for (int socket = 0; socket < FD_SETSIZE; socket++) {
            if (FD_ISSET(socket, &ready)) {
                DoSomething(socket);
            }
        }
    }

};
```

The customer found that the `FD_ISSET` always returns 0. The `FD_ISSET` macro cannot find any of the sockets that were put into it. What's going on?

Bonus chatter: People unfamiliar with the history of `select` and `fd_set` will sometimes ask, "Why do I have to pass `nfds` as the first parameter to `select` ? The system already knows the size of an `fd_set` . Why do I have to tell it?" You have to tell it because the

`fd_set` is variable-sized structure. Your program might set a custom value of `FD_SETSIZE`, and the `nfds` tells the operating system how big those bit arrays are.

I've seen arguments that `nfds` was a performance optimization, but really it was just a requirement for the ABI to know how many bits to scan. The performance optimization was a side effect.

Bonus reading: What is the maximum numeric value for a socket, and what is the maximum number of sockets a Windows program can create?

Answer to exercise: There are no macros for enumerating the contents of an `fd_set`, so this code tries to fake it by simply trying "every possible number" and seeing if it's a socket. The problem is that the numeric values of the Windows sockets created by the program are all larger than `FD_SETSIZE`, Therefore, the loop through "all possible sockets" numbered 0 through `FD_SETSIZE - 1` doesn't find them. According to the POSIX documentation for `FD_ISSET`, the file descriptor parameter to `FD_ISSET` must be a valid file descriptor; you can't just loop over "every possible number" hunting for them. The customer needs to adapt their code so they keep track of the handles which they have put in their `fd_set` and iterate over those handles (and only those handles) when calling `FD_ISSET`.

[1] If you want it to support equality comparison, you'll probably have to switch to a sorted array.

[2] And an attacker could trigger a memory exhaustion denial of service by tricking the kernel into taking a snapshot of a a ridiculously-sized bitmap.

Raymond Chen

Follow