Non-blocking I/O with regular files

Every now and them, I hear some programmer complain that a given piece of code uses blocking I/O. The claim is typically that blocking I/O damages the responsiveness of applications, especially if it has a user interface. Hence, solely non-blocking I/O should be used, along with polling (poll() or select()) or an event handling framework (glib, Qt, etc).

I can sympathize with the goal of improving applications responsiveness. **But** that is not an excuse for mixing up **blocking** with **sleeping**. Blocking is just one of several ways to sleep. In particular, non-blocking operations can sleep. Indeed turning non-blocking mode on for a file descriptor will not prevent sleeping in all possible cases, but only one (or two) of them - depending how you count.

Blocking mode refers to one specific and well-defined sort of sleep: waiting until a file descriptor can be written to or read from. What that precisely means depends on the type of the underlying file.

- For sockets, readability means there is some unread data in the *receive* buffer. That is well-known and probably the most common use case for non-blocking I/O. Conversely, writeability implies the *send* buffer is not full from the standpoint of the underlying protocol of the socket (e.g. TCP/IP). Ultimately, protocol-specific congestion control determines the exact mechanisms and policies for the send buffer.
- For pipes, readability means some unread data remains in the pipe buffer, or one task is blocking in a write to the other end of the pipe. Reciprocally, writeability means the pipe buffer has available room, or one task is blocking in a read operation on the pipe.
- FIFOs are really exactly like pipes, except that they have a name in the file system hierarchy.
- Terminals and pseudo-terminals also work much like pipes, with regards to I/O, except for the fact that they support duplex operations like sockets.
- For devices (other than terminals), polling is implementation-defined. You need to check the device driver documentation.
- For directories, reading is only defined through blocking (synchronous) function calls, notably `readdir()` or `readdir_r()`. As such polling is not defined by the specifications and it would be of no use even if it were. Also writing to directories is not allowed at all.
- Regular files are **always** readable and they are also **always** writeable. This is clearly stated in the relevant POSIX specifications. **I cannot stress this enough. Putting a regular file in non-blocking has ABSOLUTELY no effects** other than changing one bit in the file flags.

Reading from a regular file might take a long time. For instance, if it is located on a busy disk, the I/O scheduler might take so much time that the user will notice that the application is *frozen*.

Nevertheless, non-blocking mode will not fix it. It will simply not work. Checking a file for readability or writeability always succeeds immediately. If the system needs time to perform the I/O operation, it will put the task in *non-interruptible sleep* from the read or write system call. In other words, if you can assume that a file descriptor refers to a regular file, do not waste your time (or worse, other people's time) in implementing non-blocking I/O.

The only safe way to read data from or write data to a regular file while not blocking a task... consists of not performing the operation - not in that particular task anyway. Concretely, you need to create a separate thread (or process), or use asynchronous I/O (functions whose name starts with `aio_`). Whether you like it or not, and even if you think multiple threads suck, there are no other options.

An alternative, of course, involves reading small chunks of data at once, and handling other events in-between. Then again, even reading a single byte can take a long time, if said byte was not *read ahead* by the operating system.