Asynchronous I/O and event notification on linux

Updated 15/6/2008: Added proper introduction, general cleanups, made the problems with POSIX AIO clearer.
*Updated 22/9/2009:*Re-ordered the sections a bit, added information on the difference between edge- and level-triggered notification mechanisms, and added information on signalfd() and the "signal handler writes to pipe" techniques.
Updated 18/12/2017: Big overhaul, corrected some mistakes, fixed some types, removed some excessive animosity towards POSIX committee.

Introduction

"Asynchronous I/O" (or *AIO*) essentially refers to the ability of a process to perform input/output alongside its normal execution. Rather than the more common 'read (or write) and block until done' mode of operation, a program queues a read or write to be performed at some later point by the system (and is usually notified by the system when the I/O is complete).

Asynchronous I/O goes hand-in-hand with *event notification*. A program might be interested in several types of event including AIO completions, but also certain I/O readiness events which are not actually due to true AIO. A common example is a network socket; data might arrive over the network and be available for reading right away, and this constitutes an event that a program might be interested in (so that it can then read the available data).

In the latter example, event notification can allow for read and write operations to be performed both (a) without blocking normal program execution flow and (b) with a means of determining when the data has been transferred (in this case, it's transferred during the regular read or write operation). These same two features occur with AIO and for that reason, it's convenient to use the term "AIO" to describe both true Asynchronous I/O as well as event notification mechanisms which allow achieving precisely the same goals. Of course if you wanted to be a stickler you would keep these terms separate.

While AIO allows I/O to occur asynchronously with program execution, it also allows for I/O on multiple sources at one time. The tricky part of AIO is not so much queing the reads/writes as it is handling the event notifications.

Consider the case of a web server with multiple clients connected. There is one network (socket) channel and probably also one file channel for each client (the files must be read, and the data must be passed to the client over the network). One problem is, how to determine which client socket to send information to next - since, if we send on a channel whose output buffer is full, we will *block* the process and needlessly delay sending of information to other clients, some of which potentially do *not* have full output buffers. Another problem is to avoid wasting processor cycles in simply checking whether it is possible to perform I/O — to extend the web server example, if all the output buffers are full, it would be nice if the application could sleep until such time as any one of the buffers had some free space again (and be automatically woken at that time).

In general Asynchronous I/O revolves around two functions: The ability to determine that *input or output is immediately possible* without blocking or to *queue I/O operations*, and then determine that *a pending I/O operation has completed*. Both cases are examples of *asynchronous events*, that is, they can happen at any time during program execution, and the process need not actually be waiting for it to happen (though it can do so). The distinction between the two is largely a matter of operating mode (it is the difference between performing a read operation, for example, and being notified when the data is in the application's buffer, compared to simply being notified when the data is available and asking that it be copied to the application's buffer afterwards). Note however that the first case is arguably preferable since it potentially avoids a redundant copy operation (the kernel already knows where the data knows to be, and doesn't necessarily need to read it into its own buffer first).

The problem to be solved is how to receive notification of asynchronous events in a synchronous manner, so that a program can usefully deal with those events. With the exception of signals, asynchronous events do not cause any immediate execution of code within the application; so, the application must check for these events and deal with them in some way. The various AIO/event notification mechanisms discussed later provide ways to do this.

Note that I/O isn't the only thing that can happen asynchronously; unix signals can arrive, mutexes can be acquired/released, file locks can be obtained, sync() calls might complete, etc. All these things are also (at least potentially) asynchronous events that may need to be dealt with. Unfortunately the POSIX world doesn't generally recognize this; for instance there is no asynchronous version of the `fcntl(fd, F_SETLKW ...)` function.

Edge- versus level-triggered AIO mechanisms

There are several mechanism for dealing with AIO, which I'll discuss later. First, it's important to understand the difference between "edge-triggered" and "level-triggered" mechanisms.

A *level-triggered* AIO mechanism provides (when queried) information about which AIO events are still pending. In general, this translates to a set of file descriptors on which reading (or writing) can be performed without blocking.

An *edge-triggered* mechanism on the other hand provides information about which events have changed status (from non-pending to pending) since the last query.

Level-triggered mechanisms are arguably simpler to use, but in fact edge-triggered mechanisms provider greater flexibility and efficiency in certain circumstances, primarily because they do not require redundant information to be provided to the application (i.e. if the application already knows that an event is pending, it is wasteful to tell it again).

open() in non-blocking mode

It is possible to open a file (or device) in "non-blocking" mode by using the O_NONBLOCK option in the call to **open**. You can also set non-blocking mode on an already open file using the fcntl call. Both of these options are documented in the GNU libc documentation.

The result of opening a file in non-blocking mode is that calls to read() and write() will return with an error if they are unable to proceed immediately, ie. if there is no data available to read (yet) or the write buffer is full.

Non-blocking mode makes it possible to continuously iterate through the interesting file descriptors and check for available input (or check for readiness for output) simply by attempting a read (or write). This technique is called *polling* and is problematic primarily because it needlessly consumes CPU time - that is, the program *never* blocks, even when no input or output is possible on *any* file descriptor. An event notification mechanism is needed to discover when useful reads/writes are possible.

A more subtle problem with non-blocking I/O is that it generally doesn't work with regular files (this is true on linux, even when files are opened with O_DIRECT; possibly not on other operating systems). That is, opening a regular file in non-blocking mode has no effect for regular files: a read will always actually read some of the file, even if the program blocks in order to do so. In some cases this may not be important, seeing as file I/O is generally fast enough so as to not cause long blocking periods (so long as the file is local and not on a network, or a slow medium). However, it is a general weakness of the technique. In general, non-blocking I/O and the event notification mechanisms here will work with sockets and pipes, TTYs, and certain other types of device.

(Note, on the other hand, I'm not necessarily advocating that non-blocking I/O of this kind should actually be possible on regular files. The paradigm itself is flawed in this case; why should data ever be made available to read, for instance, unless there is a definite request for it and somewhere to put it? The non-blocking read itself does not serve as such a request, when considered for what it really is: two separate operations, the first being "check whether data is available" and the second being "read it if so").

As well as causing reads and writes to be non-blocking, The O_NONBLOCK flag also causes the open() call itself to be non-blocking for certain types of device (modems are the primary example in the GNU libc documentation). Unfortunately, there doesn't seem to exist a mechanism by which you can execute an open() call in a truly non-blocking manner for regular files. The only solution here is to use threads, one for each simultaneous open() operation.

It's clear that, even if non-blocking I/O were usable with regular files, it would only go part-way to solving the asynchronous I/O problem; it provides a mechanism to poll a file descriptor for data, but no mechanism for asynchronous notification of when data is available. To deal with multiple file descriptors a program would need to poll them in a loop, which is wasteful of processor time. On the other hand, when combined with one of the various mechanisms yet to be discussed, non-blocking I/O allows reading or writing of data on a file descriptor which is known to be ready up until such point as no more I/O can be performed without blocking.

It may not be strictly necessary to use non-blocking I/O when combined with a level-triggered AIO mechanism, however it is still recommended in order to avoid accidentally blocking in case you attempt more than a single read or write operation or, dare I say it, a kernel bug causes a spurious event notification.

AIO on Linux

There are several ways to deal with asynchronous events on linux; all of them presently have at least some minor problems, mainly due to limitations in the kernel.

- Threading
- Signals
- The SIGIO signal
- **select()** and **poll()** (and pselect/ppoll)
- **epoll()**
- POSIX asynchronous I/O (AIO)

Threading

The use of multiple threads is in some ways an ideal solution to the problem of asynchronous I/O, as well as asynchronous event handling in general, since it allows events to be dealt with asynchronously and any needed synchronization can be done explicitly (using mutexes and similar mechanisms).

However, for large amounts of concurrent I/O, the use of threads has significant problems for practical application due to the fact that each thread requires a stack (and therefore consumes a certain amount of memory) and the number of threads of in a process may be limited by this and other factors. Thus, it may be impractical to assign one thread to each event of interest. Also, context switching (switching between threads) incurs some not-insignificant overhead when lots of threads are involved.

Threading is presently the only way to deal with certain kinds of asynchronous operation (obtaining file locks, for example). It can potentially be combined with other types of asynchronous event handling, in order to allow performing these operations asynchronously; be warned, though, that it takes a great deal of care to get this right.

In fact, arguably the biggest argument against using threads is that is *hard*. Once you have a multi-threaded program, understanding the execution flow becomes much harder, as does debugging; and, it's entirely possibly to get bugs which manifest themselves only rarely, or only on certain machines, under certain processor loads, etc.

Signals

Signals can be sent between unix processes by using `kill()` as documented in the libc manual, or between threads using `pthread_kill()`. There are also the so-called "real-time" signal interfaces described here. Most importantly, signals can be sent automatically when certain asynchronous events occur; the details are discussed later - for now it's important to understand how signals need to be handled.

Signal handlers as an asynchronous event notification mechanism work just fine, but because they are truly executed asynchronously there is a limit to what they can usefully do (there are a limited number of C library functions which can be called safely from within a signal handler, for instance). A typical signal handler, therefore, often simply sets a flag which the program tests at prudent times during its normal execution. Alternatively, a program can use various functions available to wait for signals, rather than (or as well as) letting their handler run. These include:

- sleep(), nanosleep()
- pause()
- sigsuspend()
- sigwaitinfo(), sigtimedwait()

These functions are used only for waiting for signals (or in some cases a timeout) and can not be used to wait for other asynchronouse events. Many functions not specifically meant for waiting for signals will however return an error with errno set to `EINTR` should a signal be handled while they are executing; if relying on this, be careful to avoid race conditions — the class pattern of "enable signal, perform signal-sensitive operation, disable signal" has the issue that the signal might arrive just after the operation finishes (meaning that EINTR is not generated) but just before the signal is disabled again.

See also the discussion of `SIGIO` below.

`sigwaitinfo()` and `sigtimedwait()` are special in the above list in that they (a) avoid possible race conditions if used correctly and (b) return information about a pending signal (and remove it from the signal queue) without actually executing the signal handler. Their obvious limitation, however, is that they detect only signals and not other asynchronous events.

The SIGIO signal

File descriptors can be set to generate a signal when an I/O readiness event occurs on them - except for those which refer to regular files (which should not be surprising by now). This allows using `sleep()`, `pause()`

or `sigsuspend()` to wait for both signals and I/O readiness events, rather than using select()/poll().

The GNU libc documentation has some information on using SIGIO. It tells how you can use the `F_SETOWN` argument to `fcntl()` in order to specify which process should recieve the SIGIO signal for a given file descriptor. However, it does not mention that on linux you can also use `fcntl()` with `F_SETSIG` to specify an alternative signal, including a realtime signal. Usage is as follows:

```
    fcntl(fd, F_SETSIG, signum);
```

... where fd is the file descriptor and signum is the signal number you want to use. Setting `signum` to 0 restores the default behaviour (send `SIGIO`). Setting it to non-zero has the effect of causing the specified signal to be sent instead of `SIGIO`; if a "realtime" signal was specified, an attempt to queue it is made, and if it cannot be queued then a `SIGIO` is sent in the traditional manner. Note that non-realtime signals are not queued (i.e. only one instance can be pending) and if already pending, the specified signal is effectively merged with the pending instance (and `SIGIO` is not sent).

This technique cannot be used with regular files.

The IO signal technique is an edge-triggered machanism - A signal is sent when the I/O readiness status changes.

If a signal is successfully queued due to an I/O readiness event, additional signal handler information becomes available to advanced signal handlers (see the link on realtime signals above for more information). Specifically the handler will see `si_code` (in the `siginfo_t` structure) with one of the following values:

`POLL_IN` - data is available
`POLL_OUT` - output buffers are available (writing will not block)
`POLL_MSG` - system message available
`POLL_ERR` - input/output error at device level
`POLL_PRI` - high priority input available
`POLL_HUP` - device disconnected

Note these values are not necessarily distinct from other values used by the kernel in sending signals. So it is advisable to use a signal which is used for no other purpose. Assuming that the signal is generated to indicate an I/O event, the following two structure members will be available:

`si_band` - contains the event bits for the relevant fd, the same as would be seen using `poll()` (see discussion below)
`si_fd` - contains the relevant fd.

The IO signal technique, in conjunction with the signal wait functions, can be used to reliably wait on a set of events including both I/O readiness events and other signals. As such, it is already close to a complete solution to the problem, except that it cannot be used for regular files ("buffered asynchronous I/O") - a limitation that it shares with various other techniques yet to be discussed.

Note it is possible to assign different signals to different fd's, up to the point that you run out of signals. There is little to be gained from doing so however (it might lead to less SIGIO-yielding signal buffer overflows, but not by much, seeing as buffers are per-process rather than per-signal. I think).

If you use a real-time signal with this signal notification mechanism, you potentially have an asynchronous event handling scheme which in some cases may be more efficient than using `poll()` and perhaps even `epoll()`, which will soon be discussed. The efficiency breaks down, however, if there is a large enough number of pending notifications such that the signal queue becomes full.

Turning a signal event into an I/O event

With the I/O signal technique described above it's possible to turn an I/O readiness event on a file descriptor into a signal event; now, it's time to talk about how to do the opposite. This allows signals to be used with various other mechanisms that otherwise wouldn't allow it. Of course, you only need to do this if you don't want to resort solely to the I/O signal technique.

First, the old fashioned way. This involves creating a pipe (using the pipe() function) and having the signal handler write to one end of the pipe, thus generating data (and a readiness event) at the other end. For this to work properly, note the following:

- Writes to the pipe must be non-blocking. Otherwise, the write buffer may become full and the write operation in the signal handler will block, probably causing the whole program to hang.
- You must be prepared to correctly handle the write failing due to the write buffer being full. In general this means that information about signals can be lost (that is, a signal can be received but it may not be possible to record that it has been received). For this reason I recommend using a separate pipe per signal of interest.

The new way of converting signal events to I/O events is to use the `signalfd()` function, available from Linux kernel 2.6.22 / GNU libc version 2.8. This system call creates a file descriptor from which signal information (for specified signals) can be read directly.

The only advantage of the old technique is that it is portable, because it doesn't require the Linux-only `signalfd()` call.

The select() and poll() functions, and variants

The `select()` function is documented in the libc manual (and by POSIX). As noted, a file descriptor for a regular file is always considered ready for reading and writing; therefore, as with non-blocking I/O, select is no solution for regular files (which may be on a network or slow media).

While `select()` is interruptible by signals, it is not simple to use plain `select()` to wait for both signal and I/O readiness events without causing a race condition (see the discussion of signals above; essentially, the problem is that the signal might arrive between the time that the signal is ublocked and the time that `select()` is called).

The `pselect()` call (not documented in the GNU libc manual) allows atomically unmasking a signal and performing a select() operation (the signal mask is also restored before pselect returns); this allows waiting for one of either a specific signal or an I/O readiness event. It is possible to achieve the same thing using plain `select()` by having the signal handler generate an I/O readiness event that the `select()` call will notice and which will not be "lost" if the signal occurs outside the duration of the selection operation (see the previous section).

Behavior of the macros used to add and remove an fd from a set for use with `select` or `pselect` is undefined if given an fd greater than or equal to `FD_SETSIZE`. You should therefore always check that an fd is within the allowed limit before using it with these functions. In general Linux, and probably most POSIX-like systems, always use the lowest available file descriptor when allocating a new one, so this might not pose too much of a problem in practice, though it is potentially an annoying limitation.

Finally, select (and pselect) aren't particularly good from a performance standpoint because of the way the file descriptor sets are passed in (as a bitmask). The kernel is forced to scan the mask up to the supplied *nfds* argument in order to check which descriptors the userspace process is actually interested in (on the other hand, this means the relative efficiency increases as the number of "interesting" descriptors increases).

The `poll()` function, not documented in the GNU libc manual (but defined in POSIX and documented in the Linux man pages) is an alternative to `select()` which uses a variable sized array to hold the relevant file descriptors instead of a fixed size structure:

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

The structure `struct pollfd` is defined as:

```
struct pollfd {
    int fd;        // the relevant file descriptor
    short events;  // events we are interested in
    short revents; // events which occur will be marked here
};
```

The `events` and `revents` are bitmasks with a combination of any of the following values:

`POLLIN` - there is data available to be read
`POLLPRI` - there is urgent data to read
`POLLOUT` - writing now will not block

If the feature test macros are set for XOpen, the following are also available. Although they have different bit values, the meanings are essentially the same:

`POLLRDNORM` - data is available to be read
`POLLRDBAND` - there is urgent data to read
`POLLWRNORM` - writing now will not block
`POLLWRBAND` - writing now will not block

Just to be clear on this, when it is possible to write to an fd without blocking, all three of `POLLOUT`, `POLLWRNORM` and `POLLWRBAND` will be generated. There is no functional distinction between these values.

The following is also enabled for GNU source:

`POLLMSG` - a system message is available; this is used for dnotify and possibly other functions. If `POLLMSG` is set then `POLLIN` and `POLLRDNORM` will also be set.

... However, the Linux man page for `poll()` states that Linux "knows about but does not use" POLLMSG. I think this means that it defines the macro but does not actually return it from `poll()`.

The following additional values are not useful in `events` but may be returned in `revents`, i.e. they are implicitly polled:

`POLLERR` - an error condition has occurred
`POLLHUP` - hangup or disconnection of communications link
`POLLNVAL` - file descriptor is not open

The `nfds` argument should provide the size of the `ufds` array, and the `timeout` is specified in milliseconds.

The return from `poll()` is the number of file descriptors for which a watched event occurred (that is, an event which was set in the `events` field in the `struct pollfd` structure, or which was one of `POLLERR`, `POLLHUP` or `POLLNVAL`). The return may be 0 if the timeout was reached. The return is -1 if an error occurred, in which case `errno` will be set to one of the following:

> `EBADF` - a bad file descriptor was given
> `ENOMEM` - there was not enough memory to allocate file descriptor tables, necessary for `poll()` to function.
> `EFAULT` - the specified array was not contained in the calling process's address space.
> `EINTR` - a signal was received while waiting for events.
> `EINVAL` - if the `nfds` is ridiculously large, that is, larger than the number of fds the process is allowed to have open. Note that this implies it may be unwise to add the same fd to the listen set twice.

Note that `poll()` exhibits the same problems in waiting for signals that `select()` does. There is a `ppoll()` function in more recent kernels (2.6.16+) which changes the timeout argument to a `struct timespec *` and which adds a `sigset_t *` argument to take the desired signal mask during the wait (this function is documented in the Linux man pages). Oddly, `pselect()` is part of POSIX, but `ppoll()` is not (while `poll()` is).

The poll call is inefficient for large numbers of file descriptors, because the kernel must scan the list provided by the process each time poll is called, and the process must scan the list to determine which descriptors were active. Also, poll exhibits the same problems in dealing with regular files as *select()* does (files are considered always ready for reading, except at end-of-file, and always ready for writing).

Epoll

On newer kernels - since 2.5.45 - a new set of syscalls known as the *epoll interface* (or just *epoll*) is available. The epoll interface works in essentially the same way as `poll()`, except that the array of file descriptors is maintained in the kernel rather than userspace. Syscalls are available to create a set, add and remove fds from the set, and retrieve events from the set. This is much more efficient than traditional `poll()` as it prevents the linear scanning of the set required at both the kernel and userspace level for each `poll()` call.

```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

The `epoll_create()` function is used to create a poll set. The `size` argument is an indicator only; it doesn not limit the number of fds which can be put into the set. The return value is a file descriptor (used to identify the set) or -1 if an error occurs (the only possible error is `ENOMEM` which indicates there is not enough memory or address space to create the set in kernel space). An epoll file descriptor is deleted by calling `close()` and otherwise acts as an I/O file descriptor which has input available if an event is active on the set.

`epoll_ctl` is used to add, remove, or otherwise control the monitoring of an fd in the set donated by the first argument, `epfd`. The `op` argument specifies the operation which can be any of:

`EPOLL_CTL_ADD`
: add a file descriptor to the set. The `fd` argument specifies the fd to add. The `event` argument points to a `struct epoll_event` structure with the following members:

  `uint32_t events`
  : a bitmask of events to monitor on the fd. The values have the same meaning as for the `poll()` events, though they are named with an `EPOLL` prefix: `EPOLLIN`, `EPOLLPRI`, `EPOLLOUT`, `EPOLLRDNORM`, `EPOLLRDBAND`, `EPOLLWRNORM`, `EPOLLWRBAND`, `EPOLLMSG`, `EPOLLERR`, and `EPOLLHUP`.

    Two additional flags are possible: `EPOLLONESHOT`, which sets "One shot" operation for this fd, and `EPOLLET`, which sets edge-triggered mode (see the section on edge vs level triggered mechanisms; this flag allows epoll to act as either).

    In one-shot mode, a file descriptor generates an event only once. After that, the bitmask for the file descriptor is cleared, meaning that no further events will be generated unless `EPOLL_CTL_MOD` is used to re-enable some events. Note that this effectively also makes the notification edge-triggered, regardless of whether `EPOLLET` is also specified.

  `epoll_data_t data`
  : this is a union type which can be used to specify additional data that will be assosciated with events on the file descriptor. It has the following members:

    ```
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
    ```

`EPOLL_CTL_MOD`
: modify the settings for an existing descriptor in the set. The arguments are the same as for `EPOLL_CTL_ADD`.

`EPOLL_CTL_DEL`
: remove a file descriptor from the set. The `data` argument is ignored.

The return is 0 on success or -1 on failure, in which case `errno` is set to one of the following:

`EBADF` - the `epfd` argument is not a valid file descriptor
`EPERM` - the target fd is not supported by the epoll interface
`EINVAL` - the `epfd` argument is not an epoll set descriptor, or the operation is not supported
`ENOMEM` - there is insufficient memory or address space to handle the request

Note that, for reasons unknown, epoll does not handle regular files at all (not even by returning "always ready for read/write" as do `select()` and `poll()`) and will return `EPERM` if an attempt is made to add an fd referring to a regular file to an epoll set.

The `epoll_wait()` call is used to read events from the fd set. The `epfd` argument identifies the epoll set to check. The `events` argument is a pointer to an array of `struct epoll_event` structures (format specified above) which contain both the user data associated with a file descriptor (as supplied with `epoll_ctl()`) and the events on the fd. The size of the array is given by the `maxevents` argument. The `timeout` argument specifies the time to wait for an event, in milliseconds; a value of -1 means to wait indefinitely.

In edge-triggered mode, an event is reported only once for each time the readiness state changes from inactive to active, that is, from the sitation being absent to being present. See discussion in the section on edge vs level triggered mechanisms.

The return is 0 on success or -1 on failure, in which case `errno` is set to one of:

`EBADF` - the `epfd` argument is not a valid file descriptor
`EINVAL` - `epfd` is not an epoll set descriptor, or `maxevents` is less than 1
`EFAULT` - the memory area occupied by the specified array is not accessible with write permissions

Note that an epoll set descriptor can be used much like a regular file descriptor. That is, it can be made to generate `SIGIO` (or another signal) when input (i.e. events) is available on it; likewise it can be used with `poll()` and can even be stored inside another epoll set.

Epoll is fairly efficient compared to the poll/select variants, but it still won't work with regular files (in fact it is *harder* to use in conjunction with regular files, since it refuses to accept them at all).

Other types of event

So far we have discussed how to wait for readiness events on sockets, pipes, and certain other devices, as well as for signals.

One other type of event that a process commonly wants to deal with is child process termination. Various methods exist for this including `wait`, `waitpid` and `waitid`, but these are single-purpose methods which can't also detect I/O readiness. Fortunately, child termination generates a `SIGCHLD` signal which can be detected using the other mechanisms previously discussed. This is not particularly friendly for libraries which which to spawn and monitor subprocesses independently, however; they are forced to establish a signal handler (which may interfere with the main program or another library wishing to detect the same signal) or regularly poll the child status.

Certain other functions may block for an indefinite period of time. It would be nice to be able to execute such functions asynchronously, but in many cases it is not currently possible. Examples include:

- `open()`, `readdir()`, `stat()` and various other filesystem-related functions
- File locking

Also, as mentioned earlier, the mechanisms discussed so far do not allow for asynchronous I/O on regular files. This last one brings us to the next topic: Posix asynchronous I/O.

POSIX asynchronous I/O

The POSIX asynchronous I/O interface, which is documented in the GNU libc manual, would seem to be almost ideal for performing asynchronous I/O. After all, that's what it was designed for. But if you think that this is the case, you're in for bitter disappointment.

The documentation in the GNU libc manual (v2.3.1) is not complete — it doesn't document the "struct sigevent" structure used to control how notification of completed requests is performed. The structure has the following members:

- `int sigev_notify` - can be set to `SIGEV_NONE` (no notification), `SIGEV_THREAD` (a thread is started, executing function `sigev_notify_function`), or `SIGEV_SIGNAL` (a signal, identified by `sigev_signo`, is sent). `SIGEV_SIGNAL` can be combined with (non-POSIX) `SIGEV_THREAD_ID` in which case the signal will be delivered to a specific thread, rather than the process. The thread is identified by the `_sigev_un._tid` member - this is an obviously undocumented feature and possibly an unstable interface.
- `void (*sigev_notify_function)(sigval_t)` - if notification is done through a seperate thread, this is the function that is executed in that thread.
- `sigev_notify_attributes` - if notification is done through a seperate thread, this field specifies the attributes of that thread.

- `int sigev_signo` - if notification is to be performed by a signal, this gives the number of the signal.
- `sigval_t sigev_value` - this is the parameter passed to either the signal handler or notification function. See real-time signals for more information.

Note that in particular, "sigev_value" and "sigev_notify_attributes" are not documented in the libc manual, and the types of none of the fields is specified.

Unfortunately POSIX AIO on linux is implemented at user level, using threads! (Actually, there is an AIO implementation in the kernel. I believe it's been in there since sometime in the 2.5 series. But it may have certain limitations - see here - I've yet to ascertain current status, but I believe it's not complete, and I don't believe Glibc uses it).

In fact, the POSIX AIO API implementation in Glibc is seriously broken. Issues include:

1. It's not well explained in the Glibc manual, but partial writes/reads can occur just as with normal read()/write() calls. That's fine. You can find out how many bytes were actually read/written using aio_return(). Partial reads/writes don't really make sense for regular files but it's probably safest to assume that they can occur.
2. There is no way to use POSIX AIO to poll a socket on which you are listening for connections. It can only be used for actually reading or writing data. Ultimately, this should also be Ok because you can use `ppoll()` etc for the socket and wait for an asynchronous notification from the AIO mechanism, which is sort of ok (keep reading).
3. Of the notification methods, sending a signal would seem at the outset to be the only appropriate choice when large amounts of concurrent I/O are taking place. Although realtime signals could be used, there is a potential for signal buffer overflow which means signals could be lost; furthermore there is no notification at all of such overflow (one would think raising SIGIO in this case would be a good idea, but no, POSIX doesn't specify it, and Glibc doesn't do it). What Glibc does do is set an error on the AIO control block so that if you happen to check, you will see an error. Of course, you never *will* check because you'll never receive any notification of completion.

   Note: the POSIX standard requires that, if an AIO operation is sucessfully submitted, the specified signal *must* be able to be queued (see text here, "*... it must be possible to ensure that sufficient resources exist to deliver the signal when the event occurs*"). While this would be good from the application point of view, it would be hard to implement this in a kernel, I suspect, and the userspace implementation in Glibc has no way of achieving it at all.

4. To use AIO with signal notifications reliably then, you need to check each and every AIO control block that is associated with a particular signal whenever that signal is received. For realtime signals it means that the signal queue should be drained before this is performed, to avoid redundant checking. It would be possible to use a range of signals and distribute the control blocks to them, which would limit the amount of control blocks to check per signal received; however, it's clear that ultimately this technique is not suitable for large amounts of highly concurrent I/O.
5. The other option for notification, using threads, is clearly stupid. If you're willing to spawn a thread per AIO request you may as well just use threads as a solution to begin with, and stick to regular blocking I/O. (Yes, technically, with AIO you only get one thread per *active* channel and so potentially you need a lot less threads than you would otherwise, however, you still potentially can get a lot of threads running all at once, and they do chew up memory. Also, it's not clear what happens if it's not possible to create a new thread at the time the event occurs).
6. `aio_suspend()`, while it might seem to solve the issue of notification, requires scanning the list of the aiocb structures by the kernel (to determine whether any of them have completed) and the userspace process (to find which one completed). That is to say, it has exactly the same problems as `poll()`. Also it has the potential signal race problem discussed previously (which can be worked around by having the signal handler write to a pipe which is being monitored by the `aio_suspend` call).

In short, it's a bunch of crap.

The ideal solution

... is yet to arrive. The state of AIO support in progressive kernel versions has barely improved, leaving us only with this broken Glibc implementation.

There's occasionally talk of trying to improve the situation, but progress has been far, far slower than I'd like.

For the record, though, I think that the real solution:

- Looks more like Posix AIO than epoll. (The API could be extended to allow waiting for I/O readiness as well as completion).
- Provides a combined wait/suspend call which can wait for signals, I/O readiness events, I/O completions, and other supported asynchronous events, all at the same time, with a timeout.
- Properly handles priority, in that I/O requests should be able to have a priority assigned (The Posix AIO API does this already). When

events are polled, higher priority events should be delivered before lower priority events.
- Shares control blocks between the kernel and userspace, memory-mapped, in linked structures that avoid the need for scanning lists of block in either space. Obviously this needs a great deal of thought and planning, particularly to prevent privilege escalation holes.

The closest to this ideal that exists as far as I can tell are kqueue, from the BSD world — though it has its problems too — and, oddly enough, I/O completion ports from Microsoft Windows.

-- Davin McCall

Links and references:
Richard Gooch's I/O event handling (2002)
POSIX Asynchronous I/O for Linux - unclear whether this works with recent kernels
Buffered async IO on Jens Axboe's blog (Jan 2009)
The C10K problem by Dan Kegel. Good stuff, a bit out of date though. And what is C10K short for??
Fast UNIX Servers page by Nick Black, who informs me that C10K refers to Concurrent/Connections/Clients 10,000.

Yet to be discussed: eventfd, current kernel AIO support, syslets/threadlets, acall, timers including timerfd and setitimer, sendfile and variants, dealing with fork, multiple threads polling a single mechanism.