Syscalls: Adv. I/O 4: Nonblocking I/O

1. Overview
2. Multiplexed I/O
3. Additional Uses for I/O Muliplexing
4. **Nonblocking I/O**
   - **nonblocking mode I/O**
   - **nonblocking mode program patterns**
   - **nonblocking mode and multiplexed I/O**
   - **kernel file buffer advice**
5. Signal-Driven I/O
6. Memory-Mapped I/O
7. Zero-Copy I/O
8. Scatter-Gather I/O
9. Asynchronous I/O (AIO)

Blocking vs. Nonblocking I/O

By default, I/O occurs in **blocking mode**: a `read()` or `write()` call will **block** (i.e., cause the process to be *suspended*) until some data can be read or written.

In **nonblocking mode**, a `read()` or `write()` call *always returns immediately*:

- if it can immediately read/write data, then it does (just as in blocking mode)
- if it *cannot* immediately read/write data (so call would *block*), -1 is returned (error), and `ERRNO` is set to `EAGAIN` or `EWOULDBLOCK`

Nonblocking mode is useful when I/O can *block indefinitely.*

This can occur with pipes/FIFOs, sockets, terminals, and devices.

Nonblocking Mode and Regular Files

Note: *nonblocking mode is not relevant to I/O on* **regular files** (it has no effect).

Regular files are *automatically* **cached** (**buffered**) by the kernel when involved in I/O.

The purpose of file caching is to speed up reading/writing of files to secondary storage.

`write()` returns immediately because the data simply gets written to the **buffer cache** (**page cache**).

`read()` may have to wait on a disk read if the requested data is not in the buffer cache, but it will block only briefly.

(See "Kernel File Buffer Advice" below for how to address blocking in regular file `read()`'s.)

Setting Nonblocking Mode

There are two ways to configure an I/O connection (file descriptor) to be in **nonblocking mode**:

1. `open()` the file with the `flags` argument including `O_NONBLOCK`:
   ```
   open(file, O_...|O_NONBLOCK)
   ```
2. use `fcntl()` with `F_SETFL` to set the *file descriptor status flags* to include `O_NONBLOCK`:
   ```
   fcntl(fd, F_SETFL, O_NONBLOCK)
   ```

When successful, subsequent operations on the FD *cannot* cause the calling process to block.

Nonblocking I/O Errors Codes

When an FD is in nonblocking mode and a `read()`/`write()` is unable to immediately read/write data, -1 (error) is returned.

The calling program can determine that the error was due to inability to immediately read/write by checking `ERRNO`.

The following error codes indicate that the calls would have blocked if they were not in nonblocking mode:

- `EAGAIN` – when the FD refers to a *file other than a socket*
- `EAGAIN` or `EWOULDBLOCK` – when the FD refers to a *socket* (man pages say portable applications must check for both codes)

Using Nonblocking I/O

In nonblocking mode, I/O calls can immediately return—without performing any I/O.

It is up to the calling program to check if the I/O operation succeeded or not.

If it did not, it must arrange for the operation to be tried again.

One approach is for nonblocking I/O calls to be wrapped in a *loop* that keeps trying the operation until it is successful:

```
while ((nread = read(fd,buff,cnt)) == -1 &&
       (errno == EAGAIN || errno == EWOULDBLOCK));
if (nread == -1) ...handle true read error...
...process buff...
```

Using Nonblocking I/O (contd.)

Of course this is a **busy wait**, so potentially extremely inefficient!

Instead, one should generally use a **multiplexed I/O** mechanism (e.g., `poll()`) to keep checking if I/O is possible on the FD.

The multiplexed I/O syscalls allow a program to *efficiently* wait to call `read()`/`write()` again when the operation can succeed.

If one needs to use multiplexed I/O calls like `poll()`, though, then what is the value of nonblocking mode?

That is, if these syscalls can efficiently wait until `read()`/`write()` will succeed (not block), then there seems to be no reason to need to use nonblocking mode.

Using Nonblocking I/O (contd.)

It turns out that nonblocking mode is still valuable/necessary, because multiplexed I/O calls like `poll()` do not provide a 100% guarantee that an I/O operation won't block.

A key reason for this is because of the fact that a call like `poll()` checks at one instant, but a `read()`/`write()` happens later—when system state may have changed.

This means that code like the following could potentially block on the `read()`:

```
while (poll(fds,n,-1) > 0) {
  ...
  if (fds[i].revents & POLLIN) {
    if ((nread = read(fd,buff,cnt) == -1)
      ...handle read error...
    else
      ...process buff...
  }
  ...
}
```

Using Nonblocking I/O (contd.)

To be safe, when using multiplexed I/O calls, FDs should be set to *nonblocking mode*, and code such the following used:

```
while (poll(fds,n,-1) > 0) {
  ...
  if (fds[i].revents & POLLIN) {
    if ((nread = read(fd,buff,cnt)) == -1) {
      if (errno != EAGAIN && errno != EWOULDBLOCK)
        ...handle true read error... }
    else
      ...process buff...
  }
  ...
}
```

If `read()` fails with `EAGAIN/EWOULDBLOCK`, nothing is done with the FD on this iteration, and `poll()` simply loops again.

Nonblocking Mode and open()

Calling `open()` with `O_NONBLOCK` can alter `open()`'s behavior.

If `open()` would block due to an incompatible **file lock** or **file lease**, including `O_NONBLOCK` causes `open()` to immediately return, with error code `EWOULDBLOCK` (see "`man fcntl`").

`open()`'ing a FIFO in nonblocking mode differs from blocking mode as follows (see "`man 7 fifo`"):

- opening FIFO *read only* will succeed even if the write end has not yet been opened
- opening FIFO *write only* will fail with `ENXIO` unless the read end has already been opened

Sockets and Nonblocking Mode

Socket FDs are often set to nonblocking mode.

This is done using `fcntl()` as shown above.

Making a socket FD nonblocking has the following effects:

- `accept()` calls that would block (no pending connections), instead immediately return -1 with error `EAGAIN` or `EWOULDBLOCK`
- all input syscalls that would block (due to empty socket buffer), will return -1 with error `EAGAIN` or `EWOULDBLOCK`
- all output syscalls that would block (due to full socket buffer), will return -1 with error `EAGAIN` or `EWOULDBLOCK`
- `connect()` that cannot immediately complete connection will return -1 with error `EINPROGRESS`

Kernel File Buffer Advice

As noted earlier, I/O to *regular files* is causes the file contents to be *automatically cached/buffered* by the Linux kernel.

This is done to improve program performance and to increase disk throughput.

Linux' file buffer memory used to be called the **disk cache** but has now been merged into the general Linux **page cache**.

Nonetheless, it is common to see this memory referred to as **disk cache** or **disk buffers**.

While file caching generally works automatically, there are system calls that allow programs some control.

Kernel File Buffer Advice (contd.)

`readahead` is the simpler, Linux-specific call:
`ssize_t readahead(int fd, off64_t offset, size_t count)`

- "populates the page cache with data from a file so that subsequent reads from that file will not block on disk I/O"
- `offset` is the starting point from which data is read
- `count` specifies the number of bytes to be read
- whole pages are read, so `offset` is "rounded down" to a page boundary and bytes are read up to the next page boundary greater than or equal to `offset+count`

`readahead()` *blocks* until the specified data has been read, so it must be called in a separate thread/process to avoid blocking the program.

Kernel File Buffer Advice (contd.)

`posix_fadvise` is the POSIX standard call, which informs the kernel of an expected file access pattern, potentially allowing the kernel to optimize file access:

```
int posix_fadvise(int fd, off_t offset, off_t len, int advice)
```

- `offset` is the start of the relevant file region
- `len` is the number of bytes in the region
  (0 `len` means to end of file)
- `advice` is the "access pattern" in the region

Kernel File Buffer Advice (contd.)

Values for `advice` include:

- `POSIX_FADV_NORMAL` – no advice to give (the default value)
- `POSIX_FADV_SEQUENTIAL` - region will be accessed sequentially (from lower to higher offsets)
- `POSIX_FADV_RANDOM` – region will be accessed in random order
- `POSIX_FADV_NOREUSE` – region will be accessed only once
- `POSIX_FADV_WILLNEED` – region will be accessed soon
- `POSIX_FADV_DONTNEED` – region will not be accessed soon