Syscalls: Adv. I/O 7: Zero-Copy I/O

1. Overview
2. Nonblocking I/O
3. Multiplexed I/O
4. Additional Uses for I/O Muliplexing
5. Signal-Driven I/O
6. Memory-Mapped I/O
7. **Zero-Copy I/O**
   - **Kernel Space $\leftrightarrow$ User Space Copying**
   - **Zero-Copy I/O**
   - **sendfile()**
   - **Linux-specific splice(), etc.**
8. Scatter-Gather I/O
9. Asynchronous I/O (AIO)

Kernel Space $\leftrightarrow$ User Space Copying

(Reminder: many objects in Linux/UNIX can be considered as *"files"* in the sense that they are accessed via *file descriptors*, including regular files, pipes, sockets, etc.)

The standard method for moving/copying data from one "file" to another is with a `read()+write()` loop such as the following:

```
ssize_t nread;
while((nread = read(fd_in,buff,buffsize)) > 0)
  write(fd_out,buff,nread)  //ignoring error checking write()
```

The "problem" with this approach is that each `read()`/`write()` syscall requires that the data being transferred is *copied twice:*

- in "file" $\xrightarrow{copy}$ *kernel-space* buffer $\xrightarrow{copy}$ `buff` (*user space*)
- `buff` (*user space*) $\xrightarrow{copy}$ *kernel-space* buffer $\xrightarrow{copy}$ out "file"

Kernel Space $\leftrightarrow$ User Space Copying (contd.)

This means that the `read()+write()` loop requires every byte of data being transferred is copied *four times!*

Of course two of those copies (in and out from/to the files/devices) are unavoidable.

However, the kernel-space buffer $\leftrightarrow$ user-space buffer copy operations are not necessarily required.

(Notice also, that the `read()+write()` loop requires *four context switches* on each loop iteration.)

Zero-Copy I/O

Zero-copy operations are those where the CPU is not required to move/copy data between memory locations.

Zero-copy I/O in particular refers to I/O operations that occur solely within kernel space.

I.e., no copying of data from/to user space is needed.

Linux includes several syscalls that can transfer data between file descriptors, but are more efficient, because they do the copying *solely within kernel space*.

I.e., they implement zero-copy I/O.

This avoids the two-copies problem of `read()+write()`.

Zero-Copy I/O (contd.)

The UNIX (semi)standard zero-copy I/O syscall is `sendfile()`.

Linux also includes the Linux-specific syscalls: `splice()`, `tee()`, `vmsplice()`.

The existence of these calls might seem to indicate that one should never need to use a `read()+write()` loop.

Unfortunately, `sendfile()` has limitations on what types of files can be associated with its input FD (it no longer limits the file types of its output FD).

In particular, `sendfile()` cannot be used to transfer data *in from* a socket.

So, it is not a general purpose zero-copy I/O syscall.

sendfile()

`sendfile()` copies data from an `mmap()`'able file descriptor to another file descriptor:

```
ssize_t sendfile(int fd_out, int fd_in, off_t *offset, size_t count)
```

- `fd_out` is target FD, opened for writing
- `fd_in` is source FD, opened for reading
- `offset` determines reading position for `fd_in`:
  - if `NULL`, data read starting at the current `fd_in` offset and `fd_in` offset updated by the call
  - else, points to a variable containing offset from which to start reading data from `fd_in`, variable will be updated by call but `fd_in` offset will not
- `count` is number of bytes to copy between the file descriptors
- returns number of bytes written to `fd_out`, else -1 on error

sendfile() (contd.)

Note that because `fd_in` must support `mmap()`-like operations, *it cannot be a socket!*

In Linux kernels $\geq$2.6.33, `fd_out` can be of *any file type*.

(If `fd_out` is associated with a *regular file*, its *offset* automatically gets updated to reflect copied data.)

sendfile() (contd.)

`sendfile()` man page notes:

- The original Linux `sendfile()` was not designed to handle large file offsets, so `sendfile64()` was added; glibc's `sendfile()` wrapper function transparently deals with this.
- Applications may wish to fall back to `read()`/`write()` in the case where `sendfile()` fails with `EINVAL` or `ENOSYS`.
- If `fd_out` refers to a socket or pipe with zero-copy support, callers must ensure the transferred portions of the file referred to by `fd_in` remain unmodified until the reader on the other end of `fd_out` has consumed the transferred data
- If sending files to a TCP socket, but need to send some header data first, use the `TCP_CORK` option to minimize the number of packets sent.

Linux-Specific Zero-Copy I/O Syscalls

In 2006, additional syscalls were added to the Linux kernel to support more general zero-copy I/O.

Linus Torvalds gave clear explanations of the new calls on the LKML:

"The really high-level concept is that there is now a notion of a "random kernel buffer" that is exposed to user space.

In other words, splice() and tee() work on a kernel buffer that the user has control over, where "splice()" moves data to/from the buffer from/to an arbitrary file descriptor, while "tee()" copes the data in one buffer to another.

So in a very real (but abstract) sense, "splice()" is nothing but read()/write() to a kernel buffer, and "tee()" is a memcpy() from one kernel buffer to another."

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"Now, to get slightly less abstract, there's two important practical details:

– the "buffer" implementation is nothing but a regular old-fashioned UNIX pipe. This actually makes sense on so many levels, but mostly simply because that is exactly what a UNIX pipe has always been: it's a buffer in kernel space. That's what a pipe has always been. So the splice usage isn't conceptually anything new for pipes - it's just exposing that old buffer in a new way.

– the second part of the deal is that the buffer is actually implemented as a set of reference-counted pointers, which means that you can copy them around without actually physically copy memory. So while "tee()" from a conceptual standpoint is exactly the same as a "memcpy()" on the kernel buffer, from an implementation standpoint it really just copies the pointers and increments the refcounts....

– vmsplice() system call to basically do a "write to the buffer", but using the reference counting and VM traversal to actually fill the buffer. This means that the user needs to be careful not to re-use the user-space buffer it spliced into the kernel-space one (contrast this to "write()", which copies the actual data, and you can thus re-use the buffer immediately after a successful write), but that is often easy to do."

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"Anyway, when would you actually use a kernel buffer? Normally you'd use it it you want to copy things from one source into another, and you don't actually want to see the data you are copying, so using a kernel buffer allows you to possibly do it more efficiently, and you can avoid allocating user VM space for it (with all the overhead that implies: not just the memcpy() to/from user space, but also simply the book-keeping).

It should be noted that splice() is very much not the same as sendfile(). The buffer is really the big difference, both conceptually, and in how you actually end up using it.

A "sendfile()" call (which a lot of other OS's also implement) doesn't actually need a buffer at all, because it uses the file cache directly as the buffer it works on. So sendfile() is really easy to use, and really efficient, but fundamentally limited in what it can do.

In contrast, the whole point of splice() very much is that buffer. It means that in order to copy a file, you literally do it like you would have done it traditionally in user space:"

⇒ **standard I/O** `read()+write()` **loop code example** (not shown here!)

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"except you'd not have a buffer in user space, and the "read()" and "write()" system calls would instead be "splice()" system calls to/from a pipe you set up as your kernel buffer. But the construct would all be indentical - the only thing that changes is really where that "buffer" exists.

Now, the advantage of splice()/tee() is that you can do zero-copy movement of data, and unlike sendfile() you can do it on arbitrary data (and, as shown by "tee()", it's more than just sending the data to somebody else: you can duplicate the data and choose to forward it to two or more different users - for things like logging etc).

So while sendfile() can send files (surprise surprise), splice() really is a general "read/write in user space" and then some, so you can forward data from one socket to another, without ever copying it into user space.

Or, rather than just a boring socket->socket forwarding, you could, for example, forward data that comes from a MPEG-4 hardware encoder, and tee() it to duplicate the stream, and write one of the streams to disk, and the other one to a socket for a real-time broadcast. Again, all without actually physically copying it around in memory."

[continued on next slide]

Linux-Specific Zero-Copy I/O Syscalls (contd.)

"So splice() is strictly more powerful than sendfile(), even if it's a bit more complex to use (the explicit buffer management in the middle). That said, I think we're actually going to remove sendfile() from the kernel entirely, and just leave a compatibility system call that uses splice() internally to keep legacy users happy.

Splice really is that much more powerful a concept, that having sendfile() just doesn't make any sense except as some legacy compatibility layer around the more powerful splice()."

So, we can eliminate the kernel space $\leftrightarrow$ user space copying that occurs with a standard `read()+write()` loop, by instead doing:

```
int pfds[2]; ssize_t pipesize, nread;
pipe(pfds);
pipesize = fcntl(pfds[0],F_GETPIPE_SZ);
while((nread = splice(fd_in,NULL,pfds[1],NULL,pipesize,0)) > 0)
  splice(pfds[0],NULL,fd_out,NULL,nread,0)
```

Linux-Specific Zero-Copy I/O Syscalls (contd.)

From `splice()` man page: The three system calls...provide user-space programs with full control over an arbitrary kernel buffer, implemented within the kernel using the same type of buffer that is used for a pipe...perform the following tasks:

- `splice()` – moves data from the buffer to a file descriptor, or vice versa, or from one buffer to another.
- `tee()` – duplicates data from one buffer to another.
- `vmsplice()` – copies data from userspace memory into a buffer.

Though we talk of copying, actual copies are generally avoided. The kernel does this by implementing a pipe buffer as a set of reference-counted pointers to pages of kernel memory. The kernel creates "copies" of pages in a buffer by creating new pointers (for the output buffer) referring to the pages, and increasing the reference counts for the pages: only pointers are copied, not the pages of the buffer.

splice()

`splice()` transfers data from one file descriptor to another, where one of the FD's must refer to a *pipe:*

```
ssize_t splice(int fd_in, loff_t *offset_in, int fd_out,
               loff_t *offset_out, size_t len, unsigned int flags)
```

- `fd_in` is source FD, opened for reading
- `offset_in` determines reading position for `fd_in`
- `fd_out` is target FD, opened for writing
- `offset_out` determines writing position for `fd_out`
- `len` is number of bytes to transfer between the file descriptors
- `flags` is an options bit mask
- returns number of bytes written to `fd_out`, else -1 on error; 0 indicates no data to read and no reason to block due to no writers for `fd_in` pipe

splice() (contd.)

The following semantics apply for `fd_in` and `offset_in`:

- if `fd_in` refers to a pipe, then `offset_in` must be `NULL`;
- if `fd_in` does not refer to a pipe and `offset_in` is `NULL`, then bytes are read from `fd_in` starting from the current file offset, and the current file offset is adjusted appropriately;
- if `fd_in` does not refer to a pipe and `offset_in` is not `NULL`, then `offset_in` must point to a buffer which specifies the starting offset from which bytes will be read from `fd_in` and the file offset of `fd_in` is not changed;

Analogous statements apply for `fd_out` and `offset_out`.

tee()

`tee()` *duplicates* data from one *pipe* to another:
`ssize_t tee(int fd_in, int fd_out, size_t len, unsigned int flags)`

- `fd_in` is source pipe (read) FD
- `fd_out` is target pipe (write) FD
- `len` is number of bytes to duplicate
- `flags` is an options bit mask
- returns number of bytes written to `fd_out`, else -1 on error; 0 indicates no data to read and no reason to block due to no writers for `fd_in` pipe

Duplicates here means that `tee()` does not consume the data from `fd_in`, so that same data can be copied by a subsequent `splice()` call.

vmsplice()

`vmsplice()` transfers (possibly multiple) ranges of user-space memory into a *pipe:*

```
ssize_t vmsplice(int fd, const struct iovec *iov,
                 unsigned long nr_segs, unsigned int flags)
```

- `fd` is pipe (write) FD
- `iov` is an array of descriptions of range(s) of memory to copy
- `nr_segs` is number of ranges of user memory described by `iov` to be transferred into a pipe FD
- `flags` is an options bit mask
- returns number of bytes transferred to `fd`, else -1 on error

vmsplice() (contd.)

`iov` is an array of `iovec` structures:

```
struct iovec {
  void  *iov_base;          /* Starting address */
  size_t iov_len;           /* Number of bytes */
};
```

This is the same structure that is used with the *scatter-gather* (*vectorized*) I/O syscals such as `readv()` and `writev()`.

It is used to specify potentially multiple range of user-space memory to transfer with one syscall.

Zero-Copy IPC

Linux has also added (2015) zero-copy syscalls to transfer data between the address space of the calling *process* and the address space of another process identified by its PID.

Data moves directly between the two processes, without passing through kernel space.

This permits fast message passing, allowing messages to be exchanged with a single copy operation rather than the double copy that would be required when using, for example, shared memory or pipes.

The two syscalls are: `process_vm_readv()` and `process_vm_writev()`.

Zero-Copy IPC (contd.)

```
ssize_t process_vm_readv(pid_t pid,
                         const struct iovec *local_iov,
                         unsigned long liovcnt,
                         const struct iovec *remote_iov,
                         unsigned long riovcnt,
                         unsigned long flags)


ssize_t process_vm_writev(pid_t pid,
                          const struct iovec *local_iov,
                          unsigned long liovcnt,
                          const struct iovec *remote_iov,
                          unsigned long riovcnt,
                          unsigned long flags)
```

copy_file_range()

Linux 4.5 (March 2016) added `copy_file_range()`:

"Copying a file consists in reading the data from a file to user space memory, then copy that memory to the destination file. There is nothing wrong with this way of doing things, but it requires doing extra copies of the data to/from the process memory. In this release Linux adds a system call, `copy_file_range(2)`, which allows to copy a range of data from one file to another, avoiding the mentioned cost of transferring data from the kernel to user space and then back into the kernel.

copy_file_range() (contd.)

This system call is only very slightly faster than `cp`, because the costs of these memory copies are barely noticeable compared with the time it takes to do the actual I/O, but there are some cases where it can help a lot more. In networking filesystems such as NFS, copying data involves sending the copied data from the server to the client through the network, then sending it again from the client to the new file in the server. But with `copy_file_range(2)`, the NFS client can tell the NFS server to make a file copy from the origin to the destination file, without transferring the data over the network (for NFS, this also requires the server-side copy feature present in the upcoming NFS v4.2, and also supported experimentally in this Linux release). In next releases, local filesystems such as Btrfs, and especialized storage devices that provide copy offloading facilities, could also use this system call to optimize the copy of data, or remove some of the present limitations (currently, copy offloading is limited to files on the same mount and superblock, and not in the same file).''

copy_file_range() (contd.)

```
ssize_t copy_file_range(int fd_in, loff_t *off_in,
                        int fd_out, loff_t *off_out,
                        size_t len, unsigned int flags)
```

The following semantics apply for `off_in`, and similar statements apply to `off_out`:

- If `off_in` is `NULL`, then bytes are read from `fd_in` starting from the file offset, and the file offset is adjusted by the number of bytes copied.
- If `off_in` is not `NULL`, then `off_in` must point to a buffer that specifies the starting offset where bytes from `fd_in` will be read. The file offset of `fd_in` is not changed, but `off_in` is adjusted appropriately.