MSG_ZEROCOPY

Intro

The MSG_ZEROCOPY flag enables copy avoidance for socket send calls. The feature is currently implemented for TCP and UDP sockets.

Opportunity and Caveats

Copying large buffers between user process and kernel can be expensive. Linux supports various interfaces that eschew copying, such as sendpage and splice. The MSG_ZEROCOPY flag extends the underlying copy avoidance mechanism to common socket send calls.

Copy avoidance is not a free lunch. As implemented, with page pinning, it replaces per byte copy cost with page accounting and completion notification overhead. As a result, MSG_ZEROCOPY is generally only effective at writes over around 10 KB.

Page pinning also changes system call semantics. It temporarily shares the buffer between process and network stack. Unlike with copying, the process cannot immediately overwrite the buffer after system call return without possibly modifying the data in flight. Kernel integrity is not affected, but a buggy program can possibly corrupt its own data stream.

The kernel returns a notification when it is safe to modify data. Converting an existing application to MSG_ZEROCOPY is not always as trivial as just passing the flag, then.

More Info

Much of this document was derived from a longer paper presented at netdev 2.1. For more in-depth information see that paper and talk, the excellent reporting over at LWN.net or read the original code.

paper, slides, video
: https://netdevconf.org/2.1/session.html?debruijn

LWN article
: https://lwn.net/Articles/726917/

patchset
: [PATCH net-next v4 0/9] socket sendmsg MSG_ZEROCOPY
: https://lore.kernel.org/netdev/20170803202945.70750-1-willemdebruijn.kernel@gmail.com

Interface

Passing the MSG_ZEROCOPY flag is the most obvious step to enable copy avoidance, but not the only one.

Socket Setup

The kernel is permissive when applications pass undefined flags to the send system call. By default it simply ignores these. To avoid enabling copy avoidance mode for legacy processes that accidentally already pass this flag, a process must first signal intent by setting a socket option:

```
if (setsockopt(fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one)))
        error(1, errno, "setsockopt zerocopy");
```

Transmission

The change to send (or sendto, sendmsg, sendmmsg) itself is trivial. Pass the new flag.

```
ret = send(fd, buf, sizeof(buf), MSG_ZEROCOPY);
```

A zerocopy failure will return -1 with errno ENOBUFS. This happens if the socket option was not set, the socket exceeds its optmem limit or the user exceeds its ulimit on locked pages.

Mixing copy avoidance and copying

Many workloads have a mixture of large and small buffers. Because copy avoidance is more expensive than copying for small packets, the feature is implemented as a flag. It is safe to mix calls with the flag with those without.

Notifications

The kernel has to notify the process when it is safe to reuse a previously passed buffer. It queues completion notifications on the socket error queue, akin to the transmit timestamping interface.

The notification itself is a simple scalar value. Each socket maintains an internal unsigned 32-bit counter. Each send call with MSG_ZEROCOPY that successfully sends data increments the counter. The counter is not incremented on failure or if called with

length zero. The counter counts system call invocations, not bytes. It wraps after UINT_MAX calls.

Notification Reception

The below snippet demonstrates the API. In the simplest case, each send syscall is followed by a poll and recvmsg on the error queue.

Reading from the error queue is always a non-blocking operation. The poll call is there to block until an error is outstanding. It will set POLLERR in its output flags. That flag does not have to be set in the events field. Errors are signaled unconditionally.

```
pfd.fd = fd;
pfd.events = 0;
if (poll(&pfd, 1, -1) != 1 || pfd.revents & POLLERR == 0)
        error(1, errno, "poll");

ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
if (ret == -1)
        error(1, errno, "recvmsg");

read_notification(msg);
```

The example is for demonstration purpose only. In practice, it is more efficient to not wait for notifications, but read without blocking every couple of send calls.

Notifications can be processed out of order with other operations on the socket. A socket that has an error queued would normally block other operations until the error is read. Zerocopy notifications have a zero error code, however, to not block send and recv calls.

Notification Batching

Multiple outstanding packets can be read at once using the recvmmsg call. This is often not needed. In each message the kernel returns not a single value, but a range. It coalesces consecutive notifications while one is outstanding for reception on the error queue.

When a new notification is about to be queued, it checks whether the new value extends the range of the notification at the tail of the queue. If so, it drops the new notification packet and instead increases the range upper value of the outstanding notification.

For protocols that acknowledge data in-order, like TCP, each notification can be squashed into the previous one, so that no more than one notification is outstanding at any one point.

Ordered delivery is the common case, but not guaranteed. Notifications may arrive out of order on retransmission and socket teardown.

Notification Parsing

The below snippet demonstrates how to parse the control message: the read_notification() call in the previous snippet. A notification is encoded in the standard error format, sock_extended_err.

The level and type fields in the control data are protocol family specific, IP_RECVERR or IPV6_RECVERR.

Error origin is the new type SO_EE_ORIGIN_ZEROCOPY. ee_errno is zero, as explained before, to avoid blocking read and write system calls on the socket.

The 32-bit notification range is encoded as [ee_info, ee_data]. This range is inclusive. Other fields in the struct must be treated as undefined, bar for ee_code, as discussed below.

```
struct sock_extended_err *serr;
struct cmsghdr *cm;

cm = CMSG_FIRSTHDR(msg);
if (cm->cmsg_level != SOL_IP &&
    cm->cmsg_type != IP_RECVERR)
        error(1, 0, "cmsg");

serr = (void *) CMSG_DATA(cm);
if (serr->ee_errno != 0 ||
    serr->ee_origin != SO_EE_ORIGIN_ZEROCOPY)
        error(1, 0, "serr");

printf("completed: %u..%u\n", serr->ee_info, serr->ee_data);
```

Deferred copies

Passing flag MSG_ZEROCOPY is a hint to the kernel to apply copy avoidance, and a contract that the kernel will queue a completion notification. It is not a guarantee that the copy is elided.

Copy avoidance is not always feasible. Devices that do not support scatter-gather I/O cannot send packets made up of kernel generated protocol headers plus zerocopy user data. A packet may need to be converted to a private copy of data deep in the stack, say to compute a checksum.

In all these cases, the kernel returns a completion notification when it releases its hold on the shared pages. That notification may arrive before the (copied) data is fully transmitted. A zerocopy completion notification is not a transmit completion notification, therefore.

Deferred copies can be more expensive than a copy immediately in the system call, if the data is no longer warm in the cache. The process also incurs notification processing cost for no benefit. For this reason, the kernel signals if data was completed with a copy, by setting flag SO_EE_CODE_ZEROCOPY_COPIED in field ee_code on return. A process may use this signal to stop passing flag MSG_ZEROCOPY on subsequent requests on the same socket.

Implementation

Loopback

Data sent to local sockets can be queued indefinitely if the receive process does not read its socket. Unbound notification latency is not acceptable. For this reason all packets generated with MSG_ZEROCOPY that are looped to a local socket will incur a deferred copy. This includes looping onto packet sockets (e.g., tcpdump) and tun devices.

Testing

More realistic example code can be found in the kernel source under tools/testing/selftests/net/msg_zerocopy.c.

Be cognizant of the loopback constraint. The test can be run between a pair of hosts. But if run between a local pair of processes, for instance when run with msg_zerocopy.sh between a veth pair across namespaces, the test will not show any improvement. For testing, the loopback restriction can be temporarily relaxed by making skb_orphan_frags_rx identical to skb_orphan_frags.